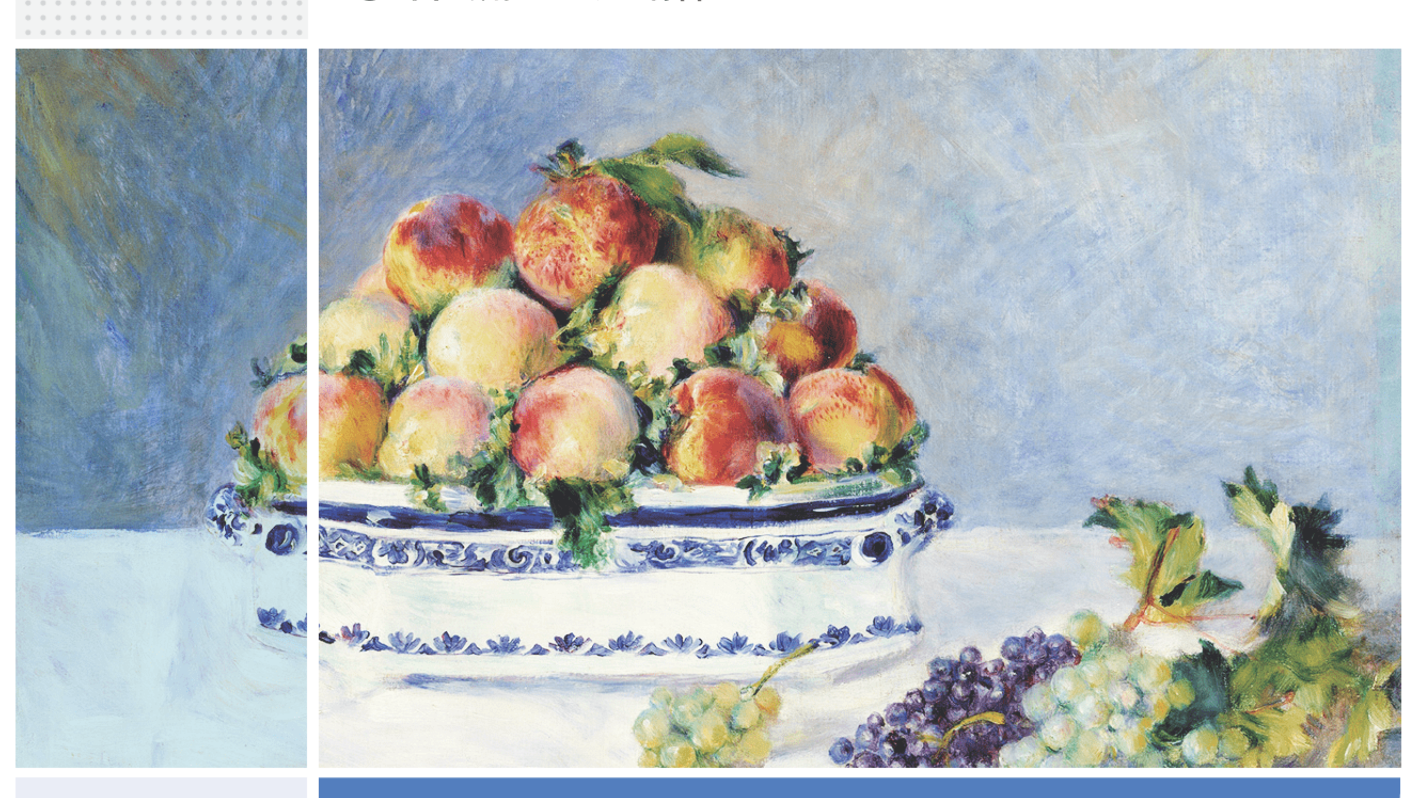


Python程序设计入门

◎ 吕云翔 孟爻 编著



零基础入门 · 案例式教学 · 微课视频讲解 · 教学资源



清華大學出版社

21世纪高等学校计算机基础实用规划教材

Python程序设计入门

吕云翔　孟　爻　编著

清华大学出版社
北　京

内 容 简 介

Python 是一种简单易学，功能强大的编程语言，它有高效率的高层数据结构，特别适用于快速的应用程序开发。全书共分为 15 章，主要内容包括 Python 简介、Python 环境搭建、Python 基础语法、函数、模块、文件操作、异常处理、面向对象编程、正则表达式、Python GUI 编程、Python 多线程与多进程编程、Python 访问数据库、Python Socket 网络编程、Python Web 编程以及 Python 综合应用实例。

本书既可以作为普通高校计算机相关专业的教材，也可以作为 Python 爱好者的参考书。

图书在版编目(CIP)数据

Python 程序设计入门/吕云翔等编著. —北京：清华大学出版社，2018
(21 世纪高等学校计算机基础实用规划教材)
ISBN 978-7-302-50147-3

Ⅰ. ①P…　Ⅱ. ①吕…　Ⅲ. ①软件工具－程序设计－高等学校－教材　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 112365 号

责任编辑：魏江江　薛　阳
封面设计：刘　键
责任校对：徐俊伟
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn,010-62795954
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　**印　张**：14.5　**字　　数**：356 千字
版　　次：2018 年 8 月第 1 版　**印　　次**：2018 年 8 月第1 次印刷
印　　数：1～1500
定　　价：39.00 元

产品编号：074945-01

前　言

Python 是一种简单易学，功能强大的编程语言，它有高效率的高层数据结构，能简单而有效地实现面向对象编程。Python 简洁的语法和对动态输入的支持，再加上解释型语言的本质，使得它在大多数平台上的很多领域都是一个理想的脚本语言，特别适用于快速的应用程序开发。

Python 可以应用于众多领域，如数据分析、组件集成、网络服务、图像处理、数值计算和科学计算等众多领域。目前业内几乎所有大中型互联网企业都在使用 Python，如 Youtube、Dropbox、BT、Quora（中国知乎）、豆瓣、知乎、Google、Yahoo!、Facebook、NASA、百度、腾讯、汽车之家、美团等。互联网公司广泛使用 Python 来做的事一般有自动化运维、自动化测试、大数据分析、爬虫、Web 等。

Python 更加易于学习和掌握，并且可利用其大量的内置函数与丰富的扩展库来快速实现许多复杂的功能。在 Python 语言的学习过程中，仍然需要通过不断的练习与体会来熟悉 Python 的编程模式，尽量不要将其他语言的编程风格用在 Python，而要从自然、简洁的角度出发，以免设计出冗长而低效率的 Python 程序。

本书的主要特色有：

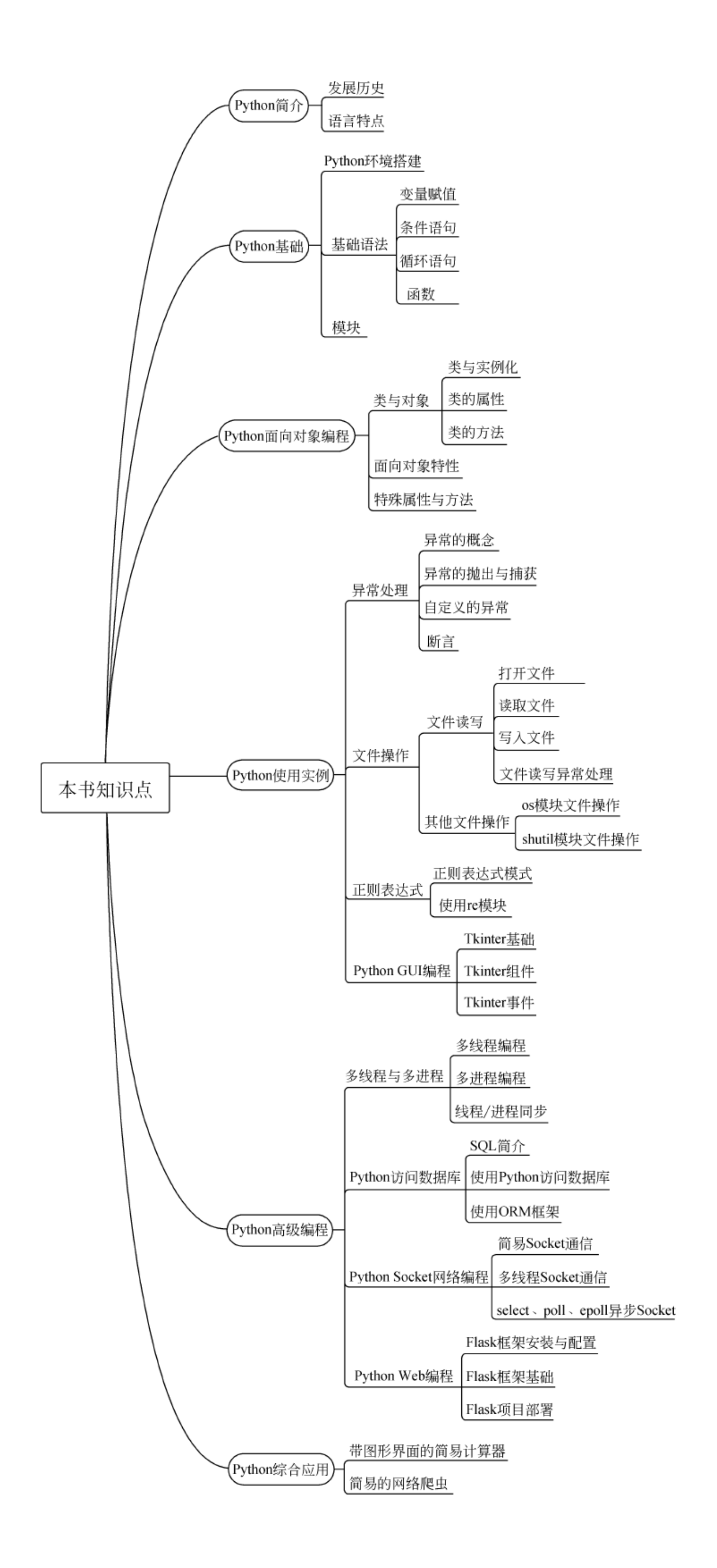
- **知识技术全面准确**：本书主要针对国内计算机相关专业的高校学生以及程序设计爱好者，书中详细介绍了 Python 语言的各种规则和规范，以便让读者能够全面掌握这门语言，从而设计出优秀的程序。
- **内容先进、体系得当**：本书的知识脉络清晰明了，第 1～4 章主要介绍 Python 的基本语法规则，第 5～9 章主要讲解一些更加深层的概念，而第 10～15 章则选取了 Python 一些在当下流行的具体应用场景下的应用。全书内容由浅入深，便于读者理解和掌握。
- **代码实例丰富完整**：对于书中每一个知识点都会配有一些示例代码并辅以相关说明文字及运行结果，还会有某些章节对一些经典的程序设计问题进行深入的讲解和探讨。读者可以参考源程序上机操作，加深体会。
- **微课辅助学习**：在某些章节，尤其是有关实际编程的章节，配有视频讲解。

本书的编著者为吕云翔、孟爻、徐祺智，另外，曾洪立、吕彼佳、姜彦华参与了部分章节的编写及配套资源制作等。

由于 Python 是一门新兴的程序设计语言，Python 语言的教学方法本身还在探索之中，加之我们的水平和能力有限，本书难免有疏漏之处，恳请各位同仁和广大读者给予批评指正，也希望各位能将实践过程中的经验和心得与我们交流（yunxianglu@hotmail.com）。

编　者

2018 年 5 月


本书知识点
Python简介
发展历史
语言特点
Python基础
Python环境搭建
基础语法
变量赋值
条件语句
循环语句
函数
模块
Python面向对象编程
类与对象
类与实例化
类的属性
类的方法
面向对象特性
特殊属性与方法
Python使用实例
异常处理
异常的概念
异常的抛出与捕获
自定义的异常
断言
文件操作
文件读写
打开文件
读取文件
写入文件
文件读写异常处理
其他文件操作
os模块文件操作
shutil模块文件操作
正则表达式
正则表达式模式
使用re模块
Python GUI编程
Tkinter基础
Tkinter组件
Tkinter事件
Python高级编程
多线程与多进程
多线程编程
多进程编程
线程/进程同步
Python访问数据库
SQL简介
使用Python访问数据库
使用ORM框架
Python Socket网络编程
简易Socket通信
多线程Socket通信
select、poll、epoll异步Socket
Python Web编程
Flask框架安装与配置
Flask框架基础
Flask项目部署
Python综合应用
带图形界面的简易计算器
简易的网络爬虫

目　　录

第1章 Python 简介

1.1 Python 的发展历程

自从 20 世纪 90 年代初 Python 语言诞生至今，它已被逐渐广泛应用于系统管理任务的处理和 Web 编程。

Python 的创始人为 Guido van Rossum。1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，作为 ABC 语言的一种继承。之所以选中 Python 作为该编程语言的名字，是因为他是一个叫 Monty Python 的喜剧团体的爱好者。

ABC 是由 Guido 参加设计的一种教学语言。就 Guido 本人看来，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，究其原因，Guido 认为是其非开放造成的。Guido 决心在 Python 中避免这一错误。同时，他还想实现在 ABC 中闪现过但未曾实现的东西。

就这样，Python 在 Guido 手中诞生了。可以说，Python 是从 ABC 发展起来，主要受到了 Modula-3(另一种相当优美且强大的语言，为小型团体所设计的)的影响。并且结合了 UNIX shell 和 C 的习惯。

1991 年，第一个 Python 编译器(也是解释器)诞生。它是用 C 语言实现的，并能够调用 C 语言的库文件。从一出生，Python 已经具有了类、函数、异常处理，包含表和词典在内的核心数据类型，及以模块为基础的拓展系统。

Python 语法很多来自 C 语言，但又受到 ABC 语言的强烈影响。来自 ABC 语言的一些规定直到今天还富有争议，例如强制缩进。但这些语法规定让 Python 容易读。另外，Python 聪明地选择服从一些惯例，特别是 C 语言的惯例，例如回归等号赋值。Guido 认为，如果“常识”上确立的东西，没有必要过度纠结。

Python 从一开始就特别在意可拓展性。Python 可以在多个层次上拓展。从高层上，用户可以直接引入. py 文件。在底层，用户可以引用 C 语言的库。Python 程序员可以快速地使用 Python 写. py 文件作为拓展模块。但当性能是考虑的重要因素时，Python 程序员可以深入底层，写 C 程序，编译为. so 文件引入到 Python 中使用。Python 就好像是使用钢结构建房一样，先规定好大的框架。而程序员可以在此框架下相当自由地拓展或更改。

最初的 Python 完全由 Guido 本人开发。Python 得到 Guido 同事的欢迎。他们迅速地反馈使用意见，并参与到 Python 的改进工作中。Guido 和一些同事构成 Python 的核心团队。他们将自己大部分的业余时间用于研究 Python。随后，Python 拓展到研究所之外。

Python 将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python 程序员可以花更多的时间用于思考程序的逻辑，而不是具体的实现细节。这一特征吸引了广大的程序员，Python 开始流行。

Python 被称为“Battery Included”，是说它以及其标准库的功能强大。这些是整个社区的贡献。Python 的开发者来自不同领域，他们将不同领域的优点带给 Python。例如 Python 标准库中的正则表达是参考 Perl，而 lambda、map、filter、reduce 等函数参考了 Lisp。Python 本身的一些功能以及大部分的标准库来自社区。Python 的社区不断扩大，进而拥有了自己的 newsgroup、网站，以及基金。从 Python 2.0 开始，Python 也从 maillist 的开发方式，转为完全开源的开发方式。社区气氛已经形成，工作被整个社区分担，Python 也获得了更加高速的发展。

到了今天，Python 的框架已经确立。Python 语言以对象为核心组织代码，支持多种编程范式，采用动态类型，自动进行内存回收。Python 支持解释运行，并能调用 C 库进行拓展。Python 有强大的标准库，由于标准库的体系已经稳定，所以 Python 的生态系统开始拓展到第三方包。这些包，如 Django、web. py、wxpython、numpy、matplotlib、PIL，将 Python 升级成了“物种”丰富的“热带雨林”。

Python 已经成为最受欢迎的程序设计语言之一。2011 年 1 月，它被 TIOBE 编程语言排行榜评为 2010 年度语言。自从 2004 年以后，Python 的使用率呈线性增长。

1.2 Python 的语言特点

Python 是一种面向对象、直译式计算机程序设计语言，这种语言的语法简洁而清晰，具有丰富和强大的类库，基本上能胜任我们平时需要的编程工作。

我们可以写一个 UNIX shell 脚本或 Windows 批处理文件完成任务，然而 shell 脚本更擅长于移动文件和修改文本数据，而不适合图形界面应用程序或游戏。我们可以写一个 C/C++/Java 的程序，但就算是一个简单的方案草案，也需要花费大量的时间。Python 更易于使用，可在 Windows、Mac OS X 和 UNIX 操作系统上使用，并会帮助用户更快速地完成工作。

Python 简单易用，但它是一种真正的编程语言，比 shell 脚本或批处理文件提供了更多的结构和对大型程序的支持。另外，Python 比起 C 提供了更多的错误检查，同时作为一门高级语言，它具有高级的内置数据类型，例如灵活的数组和字典。由于 Python 提供了更为通用的数据类型，比起 Awk 甚至 Perl，它适合更宽广的问题领域。同时，在做许多其他的事情上，Python 也不会比别的编程语言更复杂。

Python 允许用户将自己的程序分成不同的模块，可以在其他 Python 程序中重用这些模块。它配备了一个标准模块，用户可以自由使用这些标准模块作为程序的基本结构，或者作为例子开始学习 Python 编程。这些模块提供了类似文件 I/O、系统调用、网络编程，甚至像 Tkinter 的用户图形界面工具包。

Python 是一种解释型语言，它可以在程序开发期节省相当多的时间，因为它不需要编译和链接。Python 解释器可以交互地使用，这使得用户很容易体验 Python 语言的特性，以便于编写发布用的程序，或者进行自下而上的开发。它也是一个方便的桌面计算器。

Python 让程序可以写得很健壮和具有可读性，用 Python 编写的程序通常比 C、C++ 或 Java 要短得多，其原因如下：

（1）高级的数据类型使用户在一个语句中可以表达出复杂的操作。

（2）语句的组织是通过缩进而不是开始和结束括号。

（3）不需要变量或参数的声明。

Python 是可扩展的：如果用户知道用 C 写程序就很容易为解释器添加一个新的内置函数或模块，也能以最快速度执行关键操作，或者使 Python 程序能够链接到所需的二进制架构上（例如某个专用的商业图形库）。一旦真正迷上了 Python，可以将 Python 解释器连接到用 C 写的应用上，使得解释器作为这个应用的扩展或命令行语言。

由于 Python 语言的简洁性、易读性以及可扩展性，在国外用 Python 做科学计算的研究机构日益增多，一些知名大学已经采用 Python 来教授程序设计课程。例如卡耐基梅隆大学的编程基础、麻省理工学院的计算机科学及编程导论就使用 Python 语言讲授。众多开源的科学计算软件包都提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV、三维可视化库 VTK、医学图像处理库 ITK。而 Python 专用的科学计算扩展库就更多了，例如以下三个十分经典的科学计算扩展库：NumPy、SciPy 和 matplotlib，它们分别为 Python 提供了快速数组处理、数值运算以及绘图功能。因此 Python 语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

习　题　1

一、选择题

1. Python 在（　　）年的圣诞期间被荷兰人 Guido van Rossum 发明。

   A. 1988　　B. 1989　　C. 1990　　D. 1999

2. 第一个 Python 解释器于（　　）年问世。

   A. 1989　　B. 1990　　C. 1991　　D. 1999

3. Python 从（　　）版本开始完全开源。

   A. 1.5　　B. 2.0　　C. 2.7　　D. 3.0

二、填空题

1. Python 是一种________、________式的计算机程序设计语言，这种语言的语法简洁而清晰，具有丰富和强大的类库，基本上能胜任我们平时需要的编程工作。

2. Python 使用________来组织语句、代码块。

3. Python ________（是/否）需要声明变量或参数。

三、论述题

1. 查阅资料，了解 Python 在不同方向的应用以及对应的库有哪些。

2. 查阅资料，了解 Guido 发明 Python 的趣事。

第 2 章　Python 环境搭建

2.1　Python 安装

在开始编程之前，需要安装一些新软件。下面简要介绍如何下载和安装 Python。如果想直接跳到安装过程的介绍而不看详细的向导，可以直接访问 http://www.python.org/download，下载并安装 Python 的最新版本。

Python 在不同平台下的安装方式不同，我们分为 Windows、UNIX&Linux、MAC 三种平台分别介绍。

2.1.1　Windows 安装 Python

(1) 打开 Web 浏览器访问 http://www.python.org/download/。

(2) 在下载列表中选择 Window 平台安装包，包格式为：python-XYZ.msi 文件，XYZ 为我们要安装的版本号。

(3) 要使用安装程序 python-XYZ.msi，Windows 系统必须支持 Microsoft Installer 2.0 搭配使用。只要保存安装文件到本地计算机，然后运行它，看看我们的机器是否支持 MSI。Windows XP 和更高版本已经有 MSI，很多老机器也可以安装 MSI。

(4) 下载后，双击下载包，进入 Python 安装向导，安装非常简单，只需要使用默认的设置一直单击“下一步”按钮直到安装完成即可。

2.1.2　UNIX & Linux 安装 Python

目前很多 Linux 发行版如 Ubuntu 等已经预装了 Python 2.7 或 Python 3 的环境，如果没有预装，可以按照如下方法安装。

(1) 打开 Web 浏览器访问 http://www.python.org/download/。

(2) 选择适用于 UNIX/Linux 的源码压缩包。

(3) 下载及解压压缩包。

(4) 如果需要自定义一些选项修改 Modules/Setup。

(5) 执行./configure 脚本。

(6) 执行 make 命令。

(7) 执行 make install 命令。

(8) 执行以上操作后，Python 会安装在/usr/local/bin 目录中，Python 库安装在/usr/local/lib/pythonXX，XX 为我们使用的 Python 的版本号。

2.1.3　MAC 安装 Python

最近的 MAC 系统都自带有 Python 环境，我们也可以在链接 http://www.python.org/download/上下载最新版进行安装。

2.2　Windows 下环境变量的配置

以下为 Windows 10 中配置 Python 环境变量的具体步骤：

（1）首先找到 Python 的安装位置，如图 2.1 所示。（默认安装至 C 盘）

电脑 > 系统 (C:) > Python27

名称	修改日期	类型	大小
DLLs	2017/5/31 星期...	文件夹	
Doc	2017/5/31 星期...	文件夹	
include	2017/5/31 星期...	文件夹	
Lib	2017/5/31 星期...	文件夹	
libs	2017/5/31 星期...	文件夹	
Scripts	2017/5/31 星期...	文件夹	
tcl	2017/5/31 星期...	文件夹	
Tools	2017/5/31 星期...	文件夹	
LICENSE.txt	2016/12/17 星期...	文本文档	38 KB
NEWS.txt	2016/12/17 星期...	文本文档	464 KB
python.exe	2016/12/17 星期...	应用程序	27 KB
pythonw.exe	2016/12/17 星期...	应用程序	27 KB
README.txt	2016/12/3 星期...	文本文档	56 KB
w9xpopen.exe	2016/12/17 星期...	应用程序	109 KB

图 2.1　默认 Python 路径

（2）复制 Python 的路径，右击计算机图标，选择“属性”，然后进入高级系统设置中，单击环境变量，如图 2.2 所示。

（3）在系统变量中找到 path，向其中添加 Python 路径，如图 2.3 所示。

（4）检验 Python 是否装好，进入命令行，输入“python”，得到如下信息表示已经安装好，如图 2.4 所示。

2.3　Hello，Python

Python 脚本应用的开发有两种方式，一种是进入 Python 的交互环境下开发，另一种则是直接编写脚本文件。

使用 Python 交互环境开发的步骤如下。

同时按下 Win 键+R 键，然后执行 cmd 打开执行终端，输入 Python 命令，当出现：“>>>”说明成功进入，在>>>后面便可以输入想要输入的 Python 语句，例如：

（1）print 的输出方法是：print　“字符串”，如图 2.5 所示。

按 Enter 键执行后，我们就可以看到内容输出了。

（2）退出交互环境的函数：exit()，如图 2.6 所示。



图 2.2　环境变量界面

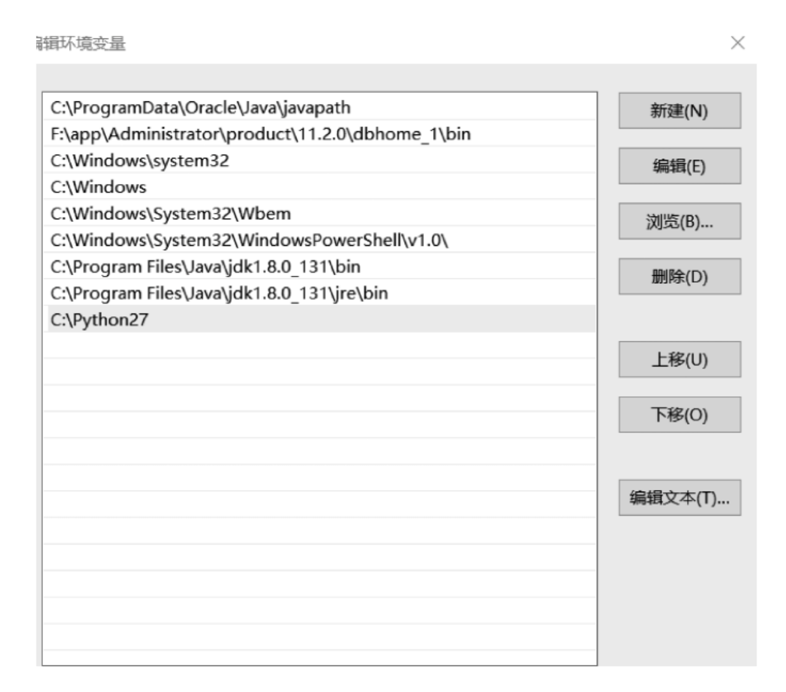


图 2.3　编辑环境变量

```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 2.4　验证 Python 安装与配置

```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello,Python'
Hello,Python
>>>
```

图 2.5　使用 print 输出字符串

```
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello,Python'
Hello,Python
>>> exit()

C:\Users\Administrator>
```

图 2.6　使用 exit()函数退出环境

可以看到我们已经退出 Python 的交互环境了。但是这种方法显得很麻烦，每次需人为输入命令，并且该命令符窗口关闭以后，前面所做的操作全部都会无效。所以，我们在实际开发的时候还是以新建脚本文件(Python 的脚本文件以.py 结尾)，然后编写该脚本，最后执行该脚本的流程学习使用，具体步骤如下：

(1) 新建一个 test.py 文件。

(2) 文本方式打开该文件(Windows 环境下建议用 notepad＋＋文本编辑器)。

(3) 输入：print 'Hello,Python'。

(4) 保存。

(5) 在同目录下新建一个 cmd.bat 文件，输入 cmd.exe 保存(该方式是快速进入工作目录)。

(6) 输入 python test.py 查看执行效果，会发现'Hello,Python'被输出，如图 2.7 所示。

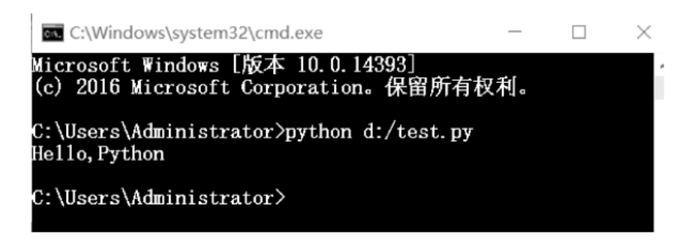


图 2.7　使用 Python 脚本文件

习　题　2

一、论述题

查阅资料,了解 Windows 环境变量中 path 的含义或 Linux 中/bin 目录的特点。

二、编程题

在 Windows 或 Linux 系统中自己搭建 Python 环境,完成“Hello World”的输出。

第3章 Python 基础语法

Python 作为一种解释型、面向对象、动态数据类型的高级程序设计语言，与 Perl、C 和 Java 等语言有许多相似之处，但也存在一些差异。本章将介绍 Python 的基础语法，让读者快速学会 Python 编程。

3.1 变量类型

变量即存储在内存中的值，这就意味着在创建变量时会在内存中开辟一个空间。基于变量的数据类型，解释器会分配指定内存，并决定什么数据可以被存储在内存中。因此，变量可以指定不同的数据类型，这些变量可以存储整数、小数或字符。

3.2 变量赋值

3.2.1 单变量赋值

Python 中的变量赋值不需要进行类型声明，每个变量都在内存中创建，包括变量的标识、名称和数据这些信息。但是每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建，其中等号(＝)用来给变量赋值。

如图 3.1(a)中所示代码分别给三个变量赋予了不同的数据类型，并输出各个变量，图 3.1(b)为输出结果。

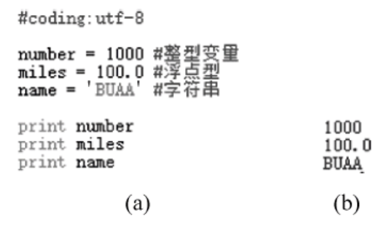


图 3.1　变量赋值实例

3.2.2 多变量赋值

Python 允许同时为多个变量赋值，如图 3.2 所示，创建一个整型对象，值为 1，三个变量被分配到相同的内存空间。另外，可以为多个对象指定多个变量，如图 3.3 所示，两个整型对象 1 和 2 分配给变量 a 和 b，字符串对象“BUAA”分配给变量 c。

```
a = b = c = 1
```

图 3.2　多变量同类型赋值

```
a,b,c = 1,2,"BUAA"
```

图 3.3　多变量不同类型赋值

3.3　数据类型

在内存中存储的数据可以有多种类型，其中 Python 定义了一些标准类型，用于存储各种类型的数据，如 Numbers（数字）、String（字符串）、List（列表）、Tuple（元组）、Dictionary（字典）等。

3.3.1　数字数据类型

数字数据类型用于存储数值，它们是不可改变的数据类型，这就意味着改变数字数据类型会分配一个新的对象。当指定数值时，Number 对象会被创建。当然，可以使用 del 语句删除一些对象的引用，语法如式(3-1)所示。

delvar1[,var2[,var3[...,varN]]]　　(3-1)

Python 支持 4 种不同类型的数字类型：int（有符号整型）、long（长整型）、float（浮点型）、complex（复数）。一些数值实例如表 3.1 所示。

表 3.1　数字数据类型的部分实例

int	long	float	complex
10	−4721885298529L	0.0	3.14j
10	5192436L	15.20	9.322e-36j
−786	0122L	70.2E-12	3e+26j

长整型也可以使用小写“l”，但还是建议使用大写“L”，避免与数字“1”混淆，其中 Python 使用“L”来显示长整型。同时，Python 还支持复数，复数由实数部分和虚数部分构成，可以用 a+bj 或者 complex(a,b)表示，复数的实部 a 和虚部 b 都是浮点型。

3.3.2　字符串数据类型

字符串或串(String)是由数字、字母、下画线组成的一串字符，它是编程语言中表示文本的数据类型。一般记为式(3-2)。

$$s = a_1 a_2 a_3 \cdots a_n \quad (n \geqslant 0) \qquad (3\text{-}2)$$

Python 的字串列表有两种取值顺序：①从左到右索引默认 0 开始的，最大范围是字符串长度少 1。②从右到左索引默认−1 开始的，最大范围是字符串开头。如果要实现从字符串中获取一段子字符串的话，可以使用变量[头下标：尾下标]，就可以截取相应的字符串，其中下标是从 0 开始算起，可以是正数或负数，下标可以为空表示取到头或尾。例如 s="ilovepython"，s[0:5]的结果是 ilove。

当使用以冒号分隔的字符串时，Python 返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。上面的结果包含了 s[0]的值 x，而取到的最大范围不包括上边界，就是 s[7]的值 d。其中加号(+)是字符串连接运算符，星号(*)是重复操作。如图 3.4 中的实例，图 3.5 为其输出结果。

```
#coding:utf-8

str = 'ilovepython'

print str           #输出完整字符串
print str[0]        #输出字符串第一个字符
print str[2:5]      #输出字符串第三至六个字符
print str[6:]       #输出字符串第七个字符（包含）之后的字符
print str*2         #输出字符串两次
print str+'really'  #输出连接字符串
```

图 3.4　字符串操作相关代码

```
ilovepython
i
ove
ython
ilovepythonilovepython
ilovepythonreally
```

图3.5　图 3.4 的输出结果

3.3.3　列表数据类型

List(列表)是 Python 中使用最频繁的数据类型,可以完成大多数集合类的数据结构实现,支持字符、数字、字符串甚至可以包含列表(即嵌套)。列表用[ ]标识,是最通用的复合数据类型,列表中值的切割也可以用变量[头下标:尾下标]截取相应的列表,从左到右索引默认 0 开始,从右到左索引默认 1 开始,下标可以为空表示取到头或尾。加号(+)是列表连接运算符,星号(*)是重复操作。相关操作代码如图 3.6 所示,图 3.7 则为其输出。

```
#coding:utf-8

list1=['BUAA',520,818,'BUPT',13.14]
tinylist = [147,'boyfriend']

print list1           #输出完整列表
print list1[0]        #输出列表第一个元素
print list1[1:3]      #输出列表第二个至第四个元素
print list1[3:]       #输出第三个元素（包含）之后的元素
print tinylist*2      #输出列表两次
print tinylist +list1#输出组合列表
```

图 3.6　列表操作相关代码

```
['BUAA', 520, 818, 'BUPT', 13.14]
BUAA
[520, 818]
['BUPT', 13.14]
[147, 'boyfriend', 147, 'boyfriend']
[147, 'boyfriend', 'BUAA', 520, 818, 'BUPT', 13.14]
```

图 3.7　图 3.6 的输出结果

3.3.4　元组数据类型

元组是一种类似于列表的数据类型,用“()”标识,内部元素用逗号隔开,但是元组不能二次赋值,相当于只读列表。相关操作代码如图 3.8 所示,图 3.9 则为其输出,另外图 3.10 表示试图更改元组元素的反馈。

3.3.5　字典数据类型

字典(Dictionary)是除列表以外 Python 之中最灵活的内置数据结构类型。列表是有序的对象集合,字典是无序的对象集合。两者之间的区别在于:字典当中的元素是通过键来存取的,而不是通过偏移来存取。

字典用“{ }”标识。字典由索引(key)和它对应的值 value 组成。图 3.11 表示了部分字典操作代码,图 3.12 则为其输出。

```
#coding:utf-8

tuple2=('BUAA',520,818,'BUPT',13.14)
tinytuple = (147,'boyfriend')

print tuple2              #输出完整元组
print tuple2[0]           #输出元组第一个元素
print tuple2[1:3]         #输出元组第二个至第四个元素
print tuple2[3:]          #输出第三个元素(包含)之后的元素
print tinytuple*2         #输出元组两次
print tinytuple+tuple2    #输出组合元组
```

图 3.8　元组操作相关代码

```
('BUAA', 520, 818, 'BUPT', 13.14)
BUAA
(520, 818)
('BUPT', 13.14)
(147, 'boyfriend', 147, 'boyfriend')
(147, 'boyfriend', 'BUAA', 520, 818, 'BUPT', 13.14)
```

图 3.9　图 3.8 的输出结果

```
>>> tuple1=(123,"BUAA",258,"BJ",520)
>>> tuple1[1]=3

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    tuple1[1]=3
TypeError: 'tuple' object does not support item assignment
```

图 3.10　试图更改元组元素

```
#coding:utf-8

dict1={}
dict1['one']="This is one"
dict1[2]="This is two"

tinydict={'name':'jeffxu','code':'818','school':'BUAA'}

print dict1['one']          #输出键为'one'的值
print dict1[2]              #输出键为2的值
print tinydict              #输出完整字典
print tinydict.keys()       #输出所有键
print tinydict.values()     #输出所有值
```

图 3.11　字典相关操作代码

```
This is one
This is two
{'school': 'BUAA', 'code': '818', 'name': 'jeffxu'}
['school', 'code', 'name']
['BUAA', '818', 'jeffxu']
```

图 3.12　图 3.11 的输出结果

3.3.6　数据类型转换

有时候,我们需要对数据内置的类型进行转换,进行数据类型的转换时,只需要将数据类型作为函数名即可,如表 3.2 所示,以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象,即转换的值。

表 3.2　部分数据转换函数

函　　数	描　　述
Int(x[,base])	将 x 转换为一个整数
Long(x[,base])	将 x 转换为一个长整数

续表

函　　数	描　　述
float(x)	将 x 转换为一个浮点数
Complex(real[,imag])	创建一个复数
Str(x)	将对象 x 转换为字符串
Repr(x)	将对象 x 转换为表达式字符串
Eval(str)	用来计算在字符串中的有效 Python 表达式,并返回一个对象
Tuple(s)	将序列 s 转换为一个元组
List(s)	将序列 s 转换为一个列表
Set(s)	转换为可变集合
Dict(d)	创建一个字典,d 必须是一个序列元组
Frozenset(s)	转换为不可变集合
Chr(x)	将一个整数转换为一个字符
Unichr(x)	将一个整数转换为 Unicode 字符
Ord(x)	将一个字符转换为其整数值
Hex(x)	将一个整数转换为一个十六进制字符串
Oct(x)	将一个整数转换为一个八进制字符串

3.4　条件语句与循环语句

3.4.1　条件语句

Python 中条件语句是通过一条或多条语句的执行结果(True 或者 False)来决定执行的代码块的,程序框图如图 3.13 所示。

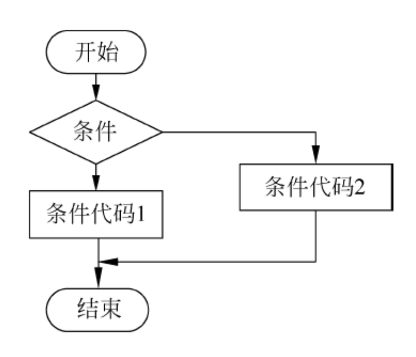


图 3.13　条件语句程序框图

Python 编程中 if 语句用于控制程序的执行,其中“判断条件”成立时(非零),则执行后面的语句,而执行内容可以多行,以缩进来区分表示同一范围。else 为可选语句,当需要在条件不成立时执行内容则可以执行相关语句。if 语句的判断条件可以用>(大于)、<(小于)、==(等于)、>=(大于等于)、<=(小于等于)来表示其关系。由于 Python 并不支持 switch 语句,所以多个条件判断,只能用 elif 来实现,如果判断需要多个条件同时判断时,可

以使用 or(或),表示两个条件有一个成立时判断条件成立;使用 and(与)时,表示只有两个条件同时成立的情况下,判断条件才成立。当 if 有多个条件时可使用括号来区分判断的先后顺序,括号中的判断优先执行,此外 and 和 or 的优先级低于>(大于)、<(小于)等判断符号,即大于和小于在没有括号的情况下会比 and 和 or 要优先判断,如图 3.14 所示,代码对于给定的数字进行判断输出,最终输出结果为 undefine。

```
#coding:utf-8

num = 8

if num < 0:     #判断是否小于0
    print 'wrong'
elif num >= 20: #判断是否大于等于20
    print 'hello'
elif (num >=0 and num <=5)or(num >=10 and num <=15): #判断是否在0~5或10~15之间
    print 'try'
else:
    print 'undefine'
```

图 3.14 if 条件语句示例

3.4.2 循环语句

程序在一般情况下是按顺序执行的,然而编程语言提供了各种控制结构,允许更复杂的执行路径。其中循环语句允许我们执行一个语句或语句组多次,图 3.15 是在大多数编程语言中的循环语句的一般格式。

其中 Python 提供了 for 循环和 while 循环,循环控制语句可以更改语句执行的顺序,Python 支持 break 语句、continue 语句以及 pass 语句,continue 用来跳过一次循环,break 用来终止循环。

for 循环语法格式如下所示,图 3.16 给出 for 循环的一个示例。

```
for 元素 in 某种元素集合:
  循环语句
```

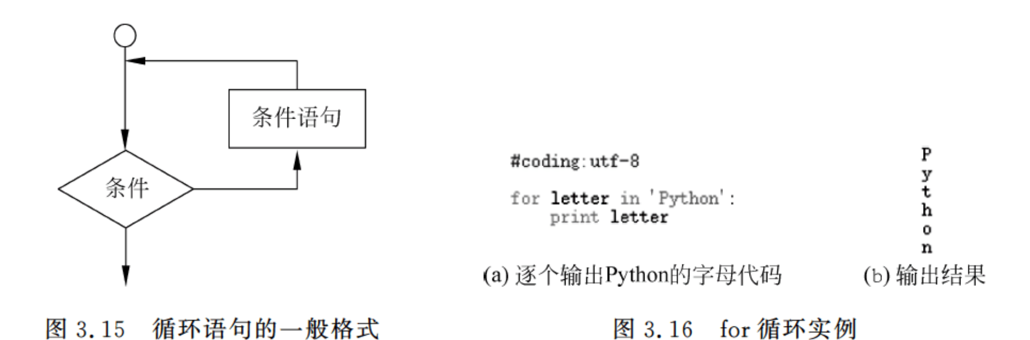


图 3.15 循环语句的一般格式　　图 3.16 for 循环实例

while 循环语法格式如下所示,图 3.17 给出 while 循环的一个示例。

```
while 循环条件:
  循环语句
```

在熟悉条件语句与循环语句的基础上,图 3.18 给出了一个二分法猜数字小游戏,采用

while 循环和条件语句实现。

```
#coding:utf-8

count = 0
while (count < 9):
   print 'The count is:', count
   count = count + 1

print "Good bye!"
```

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

(a) count加一直至count为9代码　　(b) 输出结果

图 3.17　while 循环示例

```
#coding:utf-8

import random
s = int(random.uniform(1,10))
m = int(input('input a number:'))
while m != s:
        if m > s:
                print('too big')
                m = int(input('input a number:'))
        if m < s:
                print('too small')
                m = int(input('input a number:'))
        if m == s:
                print('OK')
                break;
```

```
input a number:10
too big
input a number:5
too small
input a number:7
too big
input a number:6
OK
```

(a) 基于条件语句与while循环的代码　　(b) 输出结果

图 3.18　猜数字小游戏

习　题　3

一、选择题

1. 下列选项中,(　　)是字符串类型变量。

　A. Python　　B. 12345

　C. '12345'　　D. ("Python",)

2. 下列赋值中,(　　)是不正确的。

　A. a,b,c = 1　　B. a=b=c=1

　C. a,b,c = 1,2,3　　D. a,b,c = 1,2,"Python"

3. 有 Python 字符串变量“str = 'ILovePython'”,则“str[2:5]”的输出是(　　)。

　A. Love　　B. ILove

　C. ove　　D. Python

二、填空题

1. Python 定义了一些标准类型,用于存储各种类型的数据,其中有________,________,________,________,________。

2. Python 中,列表使用________包围,元组使用________包围,字典使用________包围。

3. Python 中,________函数用来计算字符串中有效的 Python 表达式。

三、论述题

Python 中有哪些标准类型？它们之间如何互相转换？

四、编程题

1. 某高校学生成绩按照分数划分为几段：90 分以上为 A，80 分以上为 B，70 分以上为 C，60 分以上为 D，低于 60 分为 E。编写程序，使程序输出学生分数时可以输出其对应的分数段。

2. “二分法”是编程中常用的算法之一，例如猜数字时，猜的数字是 0～100 之间的整数。首先我们取 0～100 的中间数，即 50，询问这个数比要猜的数大还是小，如果小，下一次取 50～100 的中间数，如果大，下一次取 0～50 的中间数，直到猜到正确数字。编写程序，实现二分法猜数字。

第 4 章 函　　数

4.1 函数定义

在 Python 中，定义一个函数要使用 def 语句，依次写出函数名、括号、括号中的参数和冒号“：”，然后，在缩进块中编写函数体，函数的返回值用 return 语句返回。

我们以自定义一个求绝对值的 my_abs 函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

print my_abs(1)
print my_abs(-1)
```

读者可以自行测试并调用 my_abs 看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到 return 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 return 语句，函数执行完毕后也会返回结果，只是结果为 None。另外 return None 可以简写为 return。

4.1.1 空函数

如果想定义一个什么事也不做的空函数，可以用 pass 语句，如下所示：

```
def nop():
    pass
```

pass 语句什么都不做，那有什么用？实际上 pass 可以用来作为占位符，例如现在还没想好怎么写函数的代码，就可以先放一个 pass，让代码能运行起来。

pass 还可以用在其他语句里，例如：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        pass
print my_abs(1)
print my_abs(-1)
```

缺少了 pass,代码运行时就会报语法错误。

4.1.2 参数检查

调用函数时,如果参数个数不对,Python 解释器会自动检查出来,并抛出 TypeError,如下所示:

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes exactly 1 argument (2 given)
```

但是如果参数类型不对,Python 解释器就无法帮我们检查。试试 my_abs 和内置函数 abs 的差别:

```
>>> my_abs('A')
'A'
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时,内置函数 abs 会检查出参数错误,而我们定义的 my_abs 没有参数检查,所以,这个函数定义不够完善。

让我们修改一下 my_abs 的定义,对参数类型做检查,只允许整数和浮点数类型的参数。数据类型检查可以用内置函数 isinstance 实现:

```
def my_abs(x):
    if not isinstance(x,(int,float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

添加了参数检查后,如果传入错误的参数类型,函数就可以抛出一个错误:

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续章节中讲到。

4.1.3 返回多个值

函数可以返回多个值吗？答案是肯定的。

在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

```
import math
def move(x, y, step, angle = 0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
这样我们就可以同时获得返回值：
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print x, y
151.961524227 70.0
但其实这只是一种假象，Python 函数返回的仍然是单一值：
>>> r = move(100, 100, 60, math.pi / 6)
>>> print r
(151.96152422706632, 70.0)
```

原来返回值是一个 tuple！但是，在语法上，返回一个 tuple 可以省略括号，而多个变量可以同时接收一个 tuple，按位置赋给对应的值，所以，Python 函数返回多值其实就是返回一个 tuple，但写起来更方便。

4.2 函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数和代码块结构。这个函数的基本结构完成以后，可以通过另一个函数调用执行，也可以直接从 Python 提示符执行，如下实例调用了 printme()函数：

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;
```

```
# Now you can call printme function
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
#以上实例输出结果:
#我要调用用户自定义函数!
#再次调用同一函数
```

4.2.1 按值传递参数和按引用传递参数

所有参数(自变量)在 Python 里都是按引用传递。如果我们在函数里修改了参数,那么在调用这个函数的函数里,原始的参数也被改变了。例如:

```
#!/usr/bin/python
# 可写函数说明
def changeme( mylist ):
    "修改传入的列表"
    mylist.append([1,2,3,4]);
    print "函数内取值: ", mylist
    return

# 调用 changeme 函数
mylist = [10,20,30];
changeme( mylist );
print "函数外取值: ", mylist
#传入函数的和在末尾添加新内容的对象用的是同一个引用,故输出结果如下:
#函数内取值: [10, 20, 30, [1, 2, 3, 4]]
#函数外取值: [10, 20, 30, [1, 2, 3, 4]]
```

4.2.2 函数的参数

Python 函数可以使用的参数类型如下:

- 必备参数;
- 命名参数;
- 缺省参数;
- 不定长参数。

1. 必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。调用 printme()函数,必须传入一个参数,不然会出现语法错误:

```
#!/usr/bin/python
#可写函数说明
def printme( str ):
    "打印任何传入的字符串"
    print str;
```

```
    return;

# 调用 printme 函数
printme();
# 以上实例输出结果:
# Traceback (most recent call last):
#   File "test.py", line 11, in <module>
#     printme();
# TypeError: printme() takes exactly 1 argument (0 given)
```

2. 命名参数

命名参数和函数调用关系紧密,调用方用参数的命名确定传入的参数值。我们可以跳过不传的参数或者乱序传参,因为 Python 解释器能够用参数名匹配参数值。用命名参数调用 printme()函数:

```
#!/usr/bin/python
# 可写函数说明
def printme( str ):
    "打印任何传入的字符串"
    print str;
    return;

# 调用 printme 函数
printme( str = "My string");
# 以上实例输出结果:
# My string
```

下例能将命名参数顺序不重要展示得更清楚:

```
#!/usr/bin/python
# 可写函数说明
def printinfo( name, age ):
    "打印任何传入的字符串"
    print "Name: ", name;
    print "Age ", age;
    return;

# 调用 printinfo 函数
printinfo( age = 50, name = "miki" );
# 以上实例输出结果:
# Name:  miki
# Age  50
```

3. 缺省参数

调用函数时,缺省参数的值如果没有传入,则被认为是默认值。下例会打印默认的 age,如果 age 没有被传入:

```
#!/usr/bin/python
#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用 printinfo 函数
printinfo( age=50, name="miki" );
printinfo( name="miki" );
#以上实例的输出结果:
#Name:  miki
#Age  50
#Name:  miki
#Age  35
```

4. 不定长参数

我们可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，与上述两种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号(*)的变量名会存放所有未命名的变量参数。选择不多传参数也可，如以下实例：

```
#!/usr/bin/python
# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出: "
   print arg1
   for var in vartuple:
      print var
   return;

# 调用 printinfo 函数
printinfo( 10 );
printinfo( 70, 60, 50 );
#以上实例的输出结果:
#输出:
#10
#输出:
#70
```

```
#60
#50
```

4.2.3 匿名函数

用 lambda 关键词能创建小型匿名函数。这种函数得名于省略了用 def 声明函数的标准步骤。

Lambda 函数能接收任何数量的参数但只能返回一个表达式的值，同时不能包含命令或多个表达式。匿名函数不能直接调用 print，因为 lambda 需要一个表达式。lambda 函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。

虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

lambda 函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

lambda 的使用可参考如下实例：

```
#!/usr/bin/python

#可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

#调用 sum 函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
#以上实例输出结果:
#Value of total :  30
#Value of total :  40
```

4.2.4 关于 return 语句

return 语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的 return 语句返回 None。之前的例子都没有示范如何返回数值，下例便会告诉我们该怎么做：

```
#!/usr/bin/python

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回两个参数的和."
   total = arg1 + arg2
```

```
    print "Inside the function : ", total
    return total;

# 调用 sum 函数
total = sum( 10, 20 );
print "Outside the function : ", total
#以上实例的输出结果:
#Inside the function :   30
#Outside the function :   30
```

4.2.5 变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序我们可以访问哪个特定的变量名称。在 Python 中,变量可以分为全局变量与局部变量。定义在函数内部的变量拥有一个局部作用域,定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问,而全局变量可以在整个程序范围内访问。调用函数时,所有在函数内声明的变量名称都将加入到作用域中,如以下实例:

```
#!/usr/bin/python

total = 0; # This is global variable.
# 可写函数说明
def sum( arg1, arg2 ):
    #返回两个参数的和."
    total = arg1 + arg2; # total 在这里是局部变量.
    print "Inside the function local total : ", total
    return total;

#调用 sum 函数
sum( 10, 20 );
print "Outside the function global total : ", total
#以上实例的输出结果:
#Inside the function local total :   30
#Outside the function global total :   0
```

习 题 4

一、选择题

1. 如果我们为函数预留一个空的内容,可以使用(　　)。

A. break　　B. continue　　C. pass　　D. nop

2. 如果需要使用不定长参数,参数名前需要使用(　　)修饰。

A. &　　B. $　　C. *　　D. #

3. 创建匿名函数的关键字为(　　)。

A. def　　B. pass　　C. break　　D. lambda

二、填空题

1. 在 Python 中,定义一个函数要使用________语句,依次写出________、________、________和________,然后,在________中编写函数体,函数的返回值用________语句返回。

2. 如果想定义一个什么事也不做的空函数,可以用________语句。

3. 所有参数(自变量)在 Python 里都是按________传递。如果在函数里修改了参数,那么在调用这个函数的函数里,原始的参数(是/否)被改变了。

4. 用________关键词能创建小型匿名函数。这种函数得名于省略了用 def 声明函数的标准步骤。________语句[表达式]退出函数,选择性地向调用方返回一个表达式。不带参数值的 return 语句返回________。

5. 在 Python,变量可以分为________与________。定义在________的变量拥有一个局部作用域,定义在________的变量拥有全局作用域。

三、论述题

不同语言中,函数的参数可分为按值传入或按引用传入。Python 中,函数的参数使用了哪种(些)方式?什么叫按引用传入?

四、编程题

1. 使用函数改写上一章编程题中“二分法”猜数字的习题。

2. 乘方是数学中常用的运算,n 的 m 次方是指将连续 m 个 n 相乘。请使用乘法,设计函数 pow(n,m)求 n 的 m 次方,返回 n 的 m 次方的值。

*3. 在函数中调用自身的方式称为“递归”。在“递归”的函数中,需要使用条件判断语句来终止递归。例如,在“二分法”猜数字时,我们可以使用递归来取代循环。当猜的数字错误时,我们调用自身二分函数,只需要修改猜数字的起点与终点,并将结果返回;如果猜对,直接返回中间数值即可。使用递归来完成猜数字的程序(下面给出一个简单的递归的例子,供读者参考)。

```
def count(num):
    print num
    if (num > 0):
        count(num - 1)

count(10)
#调用后,将输出从 num 到 0 的自然数
```

第5章 模　　块

5.1 模块的概念

我们来考虑如下几种场景：

(1) 编写一个 Python 程序，如果程序比较简单，则可以把代码放到一个 Python 文件中。但如果程序功能比较多，可能需要多个 Python 文件来组织源代码。而这些文件之间的代码肯定有关联，例如一个文件中的 Python 代码调用另一个 Python 文件中定义的函数，怎么能做到呢？

(2) 编写程序，肯定不会所有的东西都自己写，我们肯定会用到 Python 提供的一些标准库，那么该怎么使用呢？

(3) 我们可能想编写一个公共代码，或从外部找到一个第三方的公共代码，如何放到整个 Python 系统中？如何被自己编写的代码使用？

上面这些场景，都是在编写程序时常见的问题，而这些问题，Python 是通过模块和包的机制来解决的。

简单地说，一个模块就是一个 Python 文件，一个包包含一组模块。下面将详细说明 Python 中模块和包的概念。

5.1.1 命名空间

Python 支持面向对象编程，遵守一切皆对象的原则，所以想要好好地理解模块，一定要先理解命名空间的概念。所谓命名空间，是指标识符的可见范围。对于 Python 而言，常见的命名空间主要有以下几种：

1. Build-in Namespace（内建命名空间）

内建命名空间是任何模块均可访问的命名空间，它存放着内置的函数和异常。内建命名空间在 Python 解释器启动时创建，会一直保留，不被删除。

2. Global Namespace（全局命名空间）

每个模块拥有它自己的命名空间，叫作全局命名空间，它记录了模块的变量，包括函数、类、其他导入的模块、模块级的变量和常量。模块的全局命名空间在模块定义被读入时创建，通常模块命名空间也会一直保存到解释器退出。

3. Local Namespace（局部命名空间）

每个函数都有着自己的命名空间，叫作局部命名空间，它记录了函数的变量，包括函数的参数和局部定义的变量。当函数被调用时创建一个局部命名空间，当函数返回结果或抛

出异常时，被删除。每一个递归调用的函数都拥有自己的命名空间。

有了命名空间的概念，可以有效地解决函数或者是变量重名的问题。当一行代码要使用变量 x 的值时，Python 会到所有可用的名字空间去查找变量，按照如下顺序：

(1) 局部命名空间：特指当前函数或类的方法。如果函数定义了一个局部变量 x，或一个参数 x，Python 将使用它，然后停止搜索。

(2) 全局命名空间：特指当前的模块。如果模块定义了一个名为 x 的变量、函数或类，Python 将使用它然后停止搜索。

(3) 内置命名空间：对每个模块都是全局的。作为最后的尝试，Python 将假设 x 是内置函数或变量。

(4) 如果 Python 在这些名字空间找不到 x，它将放弃查找并引发一个 NameError 异常，如 NameError：name 'x' is not defined。

不同的命名空间中允许出现相同的函数名或者是变量名。它们彼此之间不会相互影响，例如在 Namespace A 和 Namespace B 中同时有一个名为 var 的变量，对 A. var 赋值并不会改变 B. var 的值。

5.1.2 模块

Python 中的一个模块对应的就是一个. py 文件。其中定义的所有函数或者是变量都属于这个模块。这个模块对于所有函数而言就相当于一个全局的命名空间。而每个函数又都有自己局部的命名空间。如图 5.1 所示，Graph. py 就是一个名字叫 Graph 的模块。

使用模块最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括 Python 内置的模块和来自第三方的模块，如图 5.2 所示。

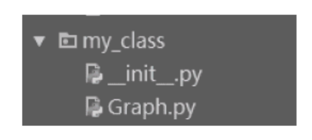


图 5.1 模块

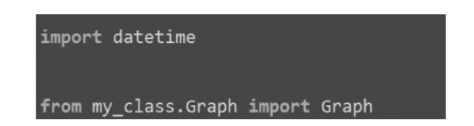


图 5.2 导入模块

如图 5.2 所示，datatime 是 Python 内置模块，我们用 import 语句导入模块后，就有了变量 datetime 指向该模块，利用 datetime 这个变量，就可以访问 datetime 模块的所有功能。而 Graph 模块此时则作为已定义模块在另一模块中的引用。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。

Python 支持以下几种导入方法：

```
import 模块  ＃导入模块
from 包/模块 import 模块/方法/对象 ＃导入包或模块下的模块或方法或对象等
import 模块 as 别名 ＃导入模块并重命名
from 包/模块 import 模块/方法/对象 as 别名
＃导入包或模块下的模块或方法或对象等并重命名
```

5.1.3 包

包是多个模块的集合，也就是多个.py 文件的集合。我们可以用如下方式来创建一个包：

（1）新建一个文件夹。

（2）在该文件夹下新建一个空的__init__.py 文件(必须存在，内容可以为空)。

（3）在该文件夹下新建 py 文件。

包提供了一种很好的管理模块的方式，可以有效地减少模块的命名冲突，不同包的模块命名可以相同，如图 5.3 所示。

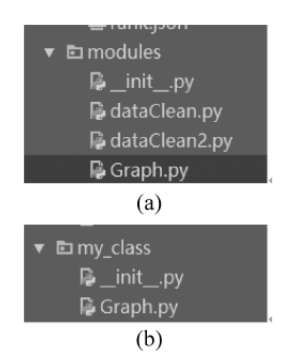


图 5.3 包的文件结构

包 modules 和包 my_class 中都有名为 Graph 的模块，但两者的内容是完全不同的，引用模块时需要带上包名，如 import my_class.Graph。

5.2 模块内置属性

在每一个模块中，都有一些内置属性，这些属性不需要手动声明或者赋值，可以直接访问。

例如，__name__为当前模块名。如果是直接运行该模块，其值为__main__；如果通过导入运行，属性值就是模块名。因此可以通过__name__属性判断该模块是直接运行还是被导入运行的，对于一些不需要在导入运行时执行的，就需要添加__name__=="__main__"判断，如下所示：

```
if __name__ == '__main__':
  print("不是 import 的")
else:
  print("是 import 的")
if __name__ == '__main__':
    print("不是 import 的")
else:
    print("是 import 的")运行结果为:
if __name__ == '__main__':
    print("不是 import 的")
else:
    print("是 import 的")
```

__file__也是 python 模块的一个内置属性，__file__用来获得模块所在的路径。__file__的返回值根据调用模块的方式不同，得到的结果可能不同。使用绝对路径调用模块时__file__将返回绝对路径，使用相对路径调用__file__时，会得到相对路径。因此，想要正确地得到绝对路径，我们需要使用 os.path.realpath(__file__)。

5.3 第三方模块安装方法

在 Python 中安装第三方模块，是通过包管理工具 pip 来完成的。如果正在使用 Windows 操作系统，在命令提示符窗口下尝试运行 pip，如果 Windows 提示未找到命令，可以重新运行安装程序添加 pip，然后再试着在命令提示符窗口下运行 pip，若仍然提示未找到命令，可以试着在命令行提示符窗口输入“python-m pip install-upgrade pip”更新 pip。

在 pip 更新完成以后，我们尝试安装第三方模块 numpy，输入命令“pip install numpy”。Numpy 是用于科学计算的 Python 库；一般来说，第三方库都会在 Python 官方的 pypi.python.org 网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者 pypi 上搜索。

安装第三方模块后，我们可以像使用自带模块一样使用它，例如：

```
>>> import numpy
>>> import sys
>>> a = numpy.arange(0,100,100)
>>> print a
```

习　题　5

一、选择题

1. 数据类型转化中，转为字符串的函数“str()”存在于(　　)命名空间中。

A. 内置命名空间　B. 全局命名空间　C. 局部命名空间　D. 包命名空间

2. 当我们使用“import datetime as dt”导入 Python 的 datetime 模块时，我们下文需要使用(　　)来引用模块。

A. datetime　B. date　C. as　D. dt

3. 通过(　　)属性可以判断当前环境是主环境还是被导入的。

A. __item__　B. __main__　C. __init__　D. __name__

二、填空题

1. Python 中命名空间有________，________，________。

2. 如果脚本不是被导入的，其__name__属性的值为________。

3. __file__属性________(是/否)总是输出绝对路径。

三、论述题

查阅资料，了解 Python 有哪些常用的自带模块、第三方模块。

四、编程题

编写一个自己的模块 mymath，该模块中需要有函数 add(a,b)、sub(a,b)、mul(a,b)、div(a,b)实现两个数的加减乘除并返回结果。

第6章　文件操作

在处理报表、实验数据或者账单时，我们往往需要从一个已有的文件中读取数据；同样，在生成报表或者账单时，我们需要向一个文件中写入数据。在任何一种编程语言中，文件操作都是一个老生常谈的话题。同样，Python 提供了非常方便的文件读写等接口。

本章，我们将介绍 Python 的文件操作。

6.1　文件读写

6.1.1　打开文件

无论是读取文件还是写入文件，我们首先都需要“打开”文件。这里的“打开”与我们使用操作系统时打开文件的意思有所区别，在写入新文件时，我们首先同样需要“打开”这个即将被创建的文件。决定是读取文件还是写入文件，取决于我们打开文件时所指定的模式。

Python 内置了 open()函数来进行文件的打开操作，它的用法如下：

```
open( 文件名 [,模式] [ , 缓冲区大小 ])
```

open 函数接受“文件名”“模式”“缓冲区大小”三个参数，其中“文件名”是必选参数，而“模式”“缓冲区大小”如果没有指定会使用默认的参数。接下来我们将详细介绍参数的使用。

“文件名”接受一个字符串类型的参数，在文件名中可以使用其相对路径或者绝对路径。

“模式”同样接受一个字符串类型的参数，它指定了对该文件采取的操作类型，如“读取”“重写”“追加”等，这些操作所对应的参数如表 6.1 所示。

表 6.1　文件打开模式

模式	描　　述
r	以只读方式打开文件。文件的指针将会放在文件的开头，这是默认模式
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头，这是默认模式
r+	打开一个文件用于读写。文件指针将会放在文件的开头
rb+	以二进制格式打开一个文件用于读写，文件指针将会放在文件的开头
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖；如果该文件不存在，创建新文件
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖；如果该文件不存在，创建新文件

续表

模式	描　　述
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖；如果该文件不存在，创建新文件
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖；如果该文件不存在，创建新文件
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾，也就是说，新的内容会写入到已有内容之后；如果该文件不存在，创建新文件进行写入
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾，也就是说，新的内容将会被写入到已有内容之后；如果该文件不存在，创建新文件进行写入
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾，文件打开时会是追加模式；如果该文件不存在，创建新文件用于读写
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾；如果该文件不存在，创建新文件用于读写

"缓冲区大小"接受一个整型参数，用来表示缓冲区的策略选择。设置为 0 时，表示不使用缓冲区，直接读写，仅在二进制模式下有效。设置为 1 时，表示在文本模式下使用行缓冲区方式。设置为大于 1 时，表示缓冲区的设置大小。如果参数 buffering 没有给出，则会使用如下默认策略：

- 对于二进制文件模式，采用固定块内存缓冲区方式，内存块的大小根据系统设备分配的磁盘块来决定，如果获取系统磁盘块的大小失败，就使用内部常量 io.DEFAULT_BUFFER_SIZE 定义的大小。一般的操作系统上，块的大小是 4096B 或者 8192B 大小。
- 对于交互的文本文件（采用 isatty()判断为 True），采用一行缓冲区的方式。其他文本文件使用与二进制一样的方式。

一般来说，如果没有特殊要求，我们使用默认的缓冲区策略即可。

open()函数会返回一个 File 对象，文件读写的后续操作都要围绕这个对象进行，我们可以使用如下方式获取该对象：

```
f = open("test.txt","w")
```

6.1.2　写入文件

在使用与"写"相关的模式打开文件后，我们便可以开始写入文件。

写入文件时，我们需要使用 File 类型对象的 write()方法：

```
file.write(内容)
```

write 接受一个字符串类型的参数。在完成文件的操作后，我们需要使用 File 类型对象的 close()方法释放文件，才能正确地访问。

我们通过一个简单的例子来展示文件写入操作：

```
f = open("test.txt","w")
f.write("Hello World!")
f.close()
```

运行这段代码，我们会在脚本文件的同一目录下得到一个内容为“Hello World!”的文本文档“text. txt”。

file. write()方法在字符串的最后不会写入换行符，因此，我们需要使用“\n”来换行：

```
f = open("test.txt","w")
f.write("Hello World!\nI love Python!")
f.close()
```

运行后，我们得到的文本文档共有两行，第一行为“Hello World!”，第二行为“I love Python!”。

在需要写入一系列字符串时，Python 还提供了更方便的方法 file. writelines(列表)，这一方法同样不会写入换行符，需要用户自己添加：

```
content = [
    "Hello World!\n",
    "I love Python!"
]

f = open("test.txt","w")
f.writelines(content)
f.close()
```

同样，运行后，我们得到的文本文档中共有两行，第一行为“Hello World!”，第二行为“I love Python!”。

6.1.3 读取文件

Python 同样为读取文件提供了非常便捷的接口。

读取文件时，我们可以使用 File 类型的 read()方法，read()方法返回一个字符串，字符串的内容就是我们读取的文件的内容。

file. read()方法还有一个可选参数“长度”，用户可以指定从文本文件中读取字符的长度。

我们使用上一节写入的文本文件进行测试，将其放在与 Python 脚本文件同一目录下（如果读者未移动其路径，此步骤可忽略）：

```
f = open("test.txt","r")
content = f.read()
print content
f.close()
```

运行后，命令行中会输出两行，分别是“Hello World!”与“I love Python!”。接下来我们测试指定读入字符长度：

```
f = open("test.txt","r")
content = f.read(5)
print content
f.close()
```

代码运行后，将会输出“Hello”。需要注意的是，当指定字符长度读取后，读取文件的游标将会移动相应的长度，也就是说，继续读取文件时，我们将得到上一次读取之后的内容：

```
f = open("test.txt","r")
print f.read(5)
print f.read(1)
print f.read(5)
f.close()
```

代码运行后，将会输出三行，分别是“Hello”“ ”“World”。

读取文件的游标可以通过 file.seek()方法设置，该方法接受一个必选参数“偏移量”和一个可选参数“起始位置”。“偏移量”即为始游标跳转到距“起始位置”多少个字符的值，“起始位置”是可选参数，默认为“0”，其可选值只有“0”“1”“2”，其含义如下：

- 0 代表从文件开头开始算起；
- 1 代表从当前位置开始算起；
- 2 代表从文件末尾算起。

例如，我们从文件中读入两次“Hello”：

```
f = open("test.txt","r")
print f.read(5)
f.seek(0)
print f.read(5)
f.close()
```

代码执行后，会输出两行“Hello”。使用 seek 方法，我们可以灵活地读取文件。除了 seek 方法，tell 也是十分实用的方法。tell 方法会返回当前游标的位置：

```
f = open("test.txt","r")
f.read(5)
print f.tell()
f.read(1)
print f.tell()
f.read(5)
print f.tell()
f.close()
```

运行后,tell 分别输出 5、6、11。

有时,我们按行读入数据,可以使用 file. readline()与 file. readlines()方法,其中,readline 方法会读取一行内容,而 readlines 会按行读取全部内容,并将读取到的行以列表的方式返回。这两个方法都会读入换行符:

```
f = open("test.txt","r")

print f.readline()
print f.readline()

f.seek(0)

print f.readlines()

f.close()
```

这段代码的输出如下:

```
Hello World!

I love Python!
['Hello World!\n', 'I love Python!']
```

6.1.4 文件读写异常处理

在进行文件操作时,很容易出现异常错误。关于一般的异常处理我们将在下一章详细讲解,本节我们将讲解文件读写的简便异常处理。

Python 为文件读写的异常处理提供了 with 语句,我们使用一个例子来简单介绍 with 的使用方法:

```
with open("test.txt","r") as f:
    print f.read()
```

with 的使用非常简单,我们只需要把打开文件的操作写在 with 后并通过 as 方法为其指定对象名即可使用。如果在 with 结构中出现异常,Python 会自动 close()文件而不会中断代码的执行;除此之外,在 with 结构中的所有代码执行完毕后,with 也会自动 close()文件,简化了文件读写。

6.2 其他文件操作

除了文件读写,Python 还提供了很多文件操作接口方便用户使用。本节我们着重介绍 os 模块与 shutil 模块下的文件操作。

在本节中,我们建立如下目录与文件进行测试:

(1) 建立 python 文件夹作为根目录；
(2) 在 python 目录下创建文本文档 text1.txt、text2.txt；
(3) 在 python 目录下建立 testDir 目录；
(4) 在 testDir 目录下创建文本文档 text3.txt。

6.2.1 os 模块文件操作

1. 当前工作路径 os.getcwd()

getcwd()返回当前 Python 脚本工作路径。我们在 python 目录下打开 cmd 或者 PowerShell 测试，如图 6.1 所示。

2. 列出文件与目录 os.listdir()

listdir()返回指定目录下所有文件名与目录名的列表，listdir()函数接受一个参数，即需要列出的路径如图 6.2 所示。

```
>>> import os
>>> os.getcwd()
'H:\\python'
```

图 6.1 当前工作路径

```
>>> import os
>>> os.listdir("H:/python")
['testDir', 'text1.txt', 'text2.txt']
>>>
```

图 6.2 列出文件与目录

3. 创建目录 os.mkdir()

mkdir()用来创建一个目录，其接受一个参数，即需要创建的目录及其路径如图 6.3 所示。

```
>>> import os
>>> os.mkdir("H:/python/newDir")
>>> os.listdir("H:/python")
['newDir', 'testDir', 'text1.txt', 'text2.txt']
```

图 6.3 创建目录

4. 重命名目录或文件 os.rename()

rename()既可以重命名文件，也可以重命名目录，其接受两个参数，分别为文件或目录的旧路径与名称、文件或目录的新路径与名称，如图 6.4 所示。

```
>>> import os
>>> os.rename("H:/python/newDir","H:/python/renameDir")
>>> os.listdir("H:/python")
['renameDir', 'testDir', 'text1.txt', 'text2.txt']
```

图 6.4 重命名目录或文件

5. 删除文件、空目录

os 模块为删除提供了多种不同的接口，在这里我们介绍两种删除方式：

- os.remove()用于删除文件而不能用于删除目录，它接受一个参数，即文件路径。
- os.rmdir()用于删除空目录，它接受一个参数，即目录路径，如果目录非空，该方法会抛出异常。

我们对这两种方法做如图 6.5 所示的测试。

```
>>> import os
>>> os.listdir("H:/python")
['renameDir', 'testDir', 'text1.txt', 'text2.txt']
>>> os.remove("H:/python/text2.txt")
>>> os.listdir("H:/python")
['renameDir', 'testDir', 'text1.txt']
>>> os.rmdir("H:/python/renameDir")
>>> os.listdir("H:/python")
['testDir', 'text1.txt']
```

图 6.5　删除文件或空目录

6. 路径相关 os. path

在 os 模块下，有一组函数均与路径相关，它们都位于 os. path 下，其中常用的函数如表 6.2 所示。

表 6.2　os. path 模块常用函数

函　　数	描　　述
os. path. exists()	判断文件是否存在，接受一个路径参数，返回布尔类型
os. path. isfile()	判断路径是否为文件，接受一个路径参数，返回布尔类型
os. path. isdir()	判断路径是否为目录，接受一个路径参数，返回布尔类型
os. path. isabs()	判断路径是否为绝对路径，接受一个路径参数，返回布尔类型
os. path. dirname()	返回路径的目录路径，接受一个路径参数
os. path. basename()	返回路径的文件名，接受一个路径参数
os. path. split()	分离目录名与文件名，接受一个路径参数，返回两个字符串，分别为目录名与文件名
os. path. splitext()	分离文件名与拓展名，接受一个路径参数，返回两个字符串，分别为文件名与拓展名
os. path. getsize()	返回文件大小，接受一个路径参数

这些函数的使用实例如图 6.6 所示。

6.2.2　shutil 模块文件操作

shutil 模块是 Python 提供的一种高层次的文件操作工具。其对文件的复制与删除操作相比于 os 模块的支持更好。

1. 复制文件 shutil. copy()

shutil. copy()用来复制两个文件，它接受两个参数，分别是需要复制的文件路径与复制后的新文件路径，如图 6.7 所示。

2. 复制目录 shutil. copytree()

shutil. copytree()用于复制目录及其内容，其接受两个参数，第一个参数是被复制的目录路径，第二个参数是复制后的目录路径，其中第二个参数中的目录必须不存在，如图 6.8 所示。

3. 移动文件或目录 shutil. move()

shutil. move()用来移动文件或目录，它接受两个参数，第一个参数是文件或目录的旧路径，第二个参数是文件或目录的新路径，如图 6.9 所示。

```
>>> import os
>>>
>>> # 测试exists()
...
>>> os.path.exists("H:/python/text1.txt")
True
>>> os.path.exists("H:/python/text4.txt")
False
>>>
>>> # 测试isfile()
...
>>> os.path.isfile("H:/python/text1.txt")
True
>>> os.path.isfile("H:/python/testDir")
False
>>>
>>> # 测试isdir()
...
>>> os.path.isdir("H:/python/text1.txt")
False
>>> os.path.isdir("H:/python/testDir")
True
>>> # 测试isabs()
...
>>> os.path.isabs("H:/python/text1.txt")
True
>>> os.path.isabs("text1.txt")
False
>>>
>>> # 测试dirname()
...
>>> os.path.dirname("H:/python/text1.txt")
'H:/python'
>>>
>>> # 测试basename()
...
>>> os.path.basename("H:/python/text1.txt")
'text1.txt'
>>>
>>> #测试split()、splitext()
...
>>> os.path.split("H:/python/text1.txt")
('H:/python', 'text1.txt')
>>> os.path.splitext("H:/python/text1.txt")
('H:/python/text1', '.txt')
>>>
>>> # 测试getsize()
...
>>> os.path.getsize("H:/python/text1.txt")
13L
```

图 6.6　os.path 模块

```
>>> import os,shutil
>>> os.listdir("H:/python")
['testDir', 'text1.txt']
>>> shutil.copy("H:/python/text1.txt","H:/python/text4.txt")
>>> os.listdir("H:/python")
['testDir', 'text1.txt', 'text4.txt']
```

图 6.7　复制文件

```
>>> shutil.copytree("H:/python/testDir","H:/python/testDir2")
>>> os.listdir("H:/python/testDir2")
['text3.txt']
```

图 6.8　复制目录

```
>>> shutil.move("H:/python/text4.txt","H:/python/testDir/text4.txt")
>>> os.listdir("H:/python/testDir")
['text3.txt', 'text4.txt']
```

图 6.9　移动文件或目录

4. 删除目录 shutil.rmtree()

os 模块中提供了删除文件与空目录的功能，而 shutil 中的 rmtree()函数提供了删除任意目录的功能，rmtree()可以删除目录与其中所有内容，如图 6.10 所示为删除目录示例。

```
>>> shutil.rmtree("H:/python/testDir2")
>>> os.listdir("H:/python")
['testDir', 'text1.txt']
```

图 6.10　删除目录

习　题　6

一、选择题

1. Python 文件读写时，可以通过(　　)结构简化异常处理。

A. try/except　　B. if/else　　C. while　　D. with/as

2. File 对象的(　　)函数用来获取读入游标的位置信息。

A. tell　　B. position　　C. seek　　D. pos

3. 如果需要重复读入内容，我们可以使用(　　)函数重定位游标。

A. tell　　B. position　　C. seek　　D. pos

二、填空题

1. Python 打开文件时，可以指定以________、________、________等常用方式打开。对文件进行写入操作后，要________才能在其他程序中正常访问。

2. 在 os 模块中，获取当前工作路径使用________方法。

3. 在 os 模块中，列出路径的文件与目录使用________方法。

4. 在 os 模块中，创建目录使用________方法。

5. 在 os 模块中，重命名文件或目录使用________方法。

三、论述题

1. 简述 Python 中读取文件、写入文件的几个常用函数以及它们的区别。

2. 简述 os.rmdir()与 shutil.rmtree()的区别。

3. 简述 os.path 模块中常用的函数及其功能。

四、编程题

一个班级的成绩单被以文本文件的方式保存，每行为一名学员的成绩。第一列为学员名，第二列为 Python 成绩，第三列为数据结构的成绩。请从文件读入计算每名学员的总分，然后将学员及他的成绩按照总分降序输出到一个新的文本文件中。示例输入文本文件内容如下：

```
Alice 100 85
Bob 90 90
Candy 70 99
David 80 85
Eason 95 85
```

第 7 章　异 常 处 理

7.1　异 常 概 念

异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行，一般情况下，在 Python 无法正常处理程序时就会发生一个异常。当 Python 脚本发生异常时我们需要捕获处理它，否则程序会终止执行。

同时，异常也表示 Python 中的一种对象，当一种异常事件发生时，Python 会产生一种对应类型的对象用来保存错误信息等。异常对象的类型既可以自己定义，也可以使用已有的异常类型。Python 提供 Python 标准异常作为一般的异常类型。

表 7.1 列出了 Python 标准异常。

表 7.1　标准异常

异 常 名 称	描　　述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器发生异常通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除零
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入，到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类

续表

异常名称	描述
IndexError	序列中没有此索引
KeyError	映射中没有这个键
MemoryError	内存溢出错误
NameError	未声明/初始化对象
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用用试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时的错误
UnicodeTranslateError	Unicode 转换时的错误

在 Python 中，所有异常都继承于基类 Exception（继承的概念会在第 8 章 Python 面向对象中详细讲解，在此读者可以理解为所有标准异常都是 Exception 指定了具体异常类型后的结果）。

7.2 异常的抛出与捕获

在了解了异常的概念后，我们来学习如何处理异常。

在进行异常处理时，我们常用 try/except 语句，用来监听 try 语句中的异常，从而让 except 语句捕获异常信息并处理。

```
try:
<语句>#需要检测异常的代码
except <名字>:
<语句>      #如果在 try 部分引发了<名字>异常，则执行
except <名字>,<数据>:
<语句>      #如果引发<名字>异常，则执行，并获得附加数据<数据>
else:
<语句>      #如果没有异常处理(如不需要特殊处理可省略)
```

try 的工作原理是，当开始一个 try 语句后，Python 就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try 子句先执行，接下来会发生什么依赖于执行时是否出现异常。如果 try 后的语句执行时发生异常，Python 就跳回到 try 并执行第一个匹配该

异常的 except 子句,异常处理完毕,控制流就通过整个 try 语句(除非在处理异常时又引发新的异常)。如果在 try 后的语句里发生了异常,却没有匹配的 except 子句,异常将被递交到上层的 try,或者到程序的最上层(这样将结束程序,并打印默认的出错信息)。如果在 try 子句执行时没有发生异常,Python 将执行 else 语句后的语句(如果有 else 的话),然后控制流通过整个 try 语句。

如图 7.1 所示给出了异常处理的一个示例。

```
#coding:utf-8

def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "No numbers\n", Argument

temp_convert("xyz");
```

(a) 代码

```
No numbers
invalid literal for int() with base 10: 'xyz'
```

(b) 输出结果

图 7.1　异常处理

在这个例子中,try 部分的代码视图将字符串"xyz"转换成 int 类型,这句代码引发了标准异常中 ValueError 类型异常。在捕获异常时,我们得到 ValueError 类型对象 Argument 并将错误信息与 Argument 携带的信息一同通过 print 语句打印,得到如图 7.1 所示的输出。

需要注意的是,捕获异常时,可以通过捕获基类类型的异常捕获所有其子类的异常类型(基类、子类的概念详见第 8 章,在此可理解为父子的关系)。在上一节中,我们介绍了所有 Python 的异常都集成于 Exception 类型,因此如果不需要捕获特定类型的异常,我们只需要“except Exception:”即可捕获所有异常,保证后续代码的正确执行。

7.3　自定义异常

在开发一个新功能的模块时,开发者经常需要自定义新类型异常来方便其他开发者使用该模块。幸运的是,Python 也提供了十分简便的自定义异常的方式。

在 Python 中自定义异常,我们只需要新建一个类型并继承 Exception 即可(关于继承的概念详见第 8 章,为了本章知识的连贯性与完整性,我们假设读者对继承的概念有所了解,没有基础的读者可以跳过本节,在学习面向对象编程之后再来学习本节的知识)。在需要抛出异常的位置,使用 raise 语句即可。在使用时,我们只需要使用 try 语句包围 raise 抛出异常的函数并在 expect 中捕获对应的自定义类型的异常,就可以像使用标准异常一样使用。

我们通过下面的例子体会自定义异常的使用:

```
class DivByZeroError(Exception):
    def __init__(self,a,b):
```

```
        self.a = a
        self.b = b
        Exception.__init__(self,str(self))
    def __str__(self):
        return "%s is divided by %s" % (self.a,self.b)

def div(a,b):
    if b != 0:
        return a/b
    else:
        raise DivByZeroError(a,b)
try:
    print div(9,3)
except DivByZeroError,e:
    print e

try:
    print div(9,0)
except DivByZeroError,e:
    print "Error:",e
```

这段代码的功能是编写了整除函数 div(),使被除数在被 0 除时抛出我们自定义的异常类型 DivByZeroError。

这段代码看上去比较晦涩,我们将其分成三部分来理解。

第一部分,我们使用 class 自定义了 DivByZeroError 异常,并重写了它的初始化函数,在构造时,我们传入两个参数,分别是被除数 a 与除数 b,我们将其保存为该类的属性,方便之后调用,在构造的最后我们调用了父类 Exception 的构造方法完成自定义类的构造;除此之外,我们还重写了其__str__方法,该方法在将该类型的对象转化为字符串时调用,例如在构造方法中,我们父类初始化函数第二个参数是错误信息,我们直接传入了字符串化后的当前对象,该类对象字符串化后为“被除数 is devided by 除数”。

第二部分是我们编写的函数 div(),在 div 中,我们判断了除数是否为 0,如果除数为 0,我们使用 raise 抛出了自定义类型 DivByZeroError 的异常对象,同时在构造该对象时传入被除数与除数。

第三部分是代码最后的两个 try/except 结构,在这个结构中是我们熟悉的异常处理流程。在第一个 try/except 中,我们计算 9/3,不引起异常,我们将输出正确结果 3;而第二个 try/except 中,我们试图计算 9/0,在 div 函数中检测到除数为 0 时,将抛出 DivByZeroError 异常,被我们的 except 语句接受,因此我们将执行 except 部分的代码:输出“Error:”与接受到的异常对象 e。

这段代码运行的结果如下:

```
3
Error: 9 is divided by 0
```

7.4 使用断言异常处理

断言常用于单元测试，其语法十分简洁：

```
assert 条件，"错误信息"
```

当断言的条件返回 True 时，程序会继续执行；当条件返回 False 时，程序会抛出 AssertionError 并输出其后的错误信息。

例如：

```
assert 1 > 0,"no"
assert 1 < 0,"no"

'''
该程序执行到第二句时会中断并抛出异常:AssertionError: no
'''
```

assert 在测试代码时十分方便，但是请不要在非测试的代码中使用断言。断言在开启"-O"的编译后(Python 模块代码会被编译成 bytecode 提高运行速度)不会被执行，因此，一般情况下，我们只在进行测试时使用断言。

习 题 7

一、选择题

1. Python 异常都基于基类(　　)。

A. Exception　　B. Error　　C. Base　　D. Try

2. 自定义异常时，我们使用(　　)关键字抛出异常。

A. try　　B. except　　C. raise　　D. catch

3. 当断言(　　)时，会抛出断言异常。

A. 成立　　B. 不成立　　C. 不确定　　D. 返回 None

二、填空题

1. 异常即是一个________，该________会在程序执行过程中发生，影响了程序的正常执行，一般情况下，在 Python 无法正常处理程序时就会________。

2. 在 Python 中，所有异常都继承于基类________。

3. 在进行异常处理时，我们常用________语句，它用来监听________语句中的异常，从而让________语句捕获异常信息并处理。

4. 在 Python 中自定义异常，我们只需要新建一个________并继承________即可。在需要抛出异常的位置，使用________语句即可。在使用时，我们只需要使用________语句包围________并在________中捕获对应的自定义类型的异常。

三、论述题

1．哪些场合适合使用断言？哪些场合不适合？为什么？

2．简述 try、except、raise 之间的关系与它们的作用。

四、编程题

编写自定义异常 DivError 类，用于自定义整除 div 函数抛出异常。DivError 类所包含的异常信息有除数、被除数、异常原因（a. 除数或被除数中有非整数；b. 除数或被除数中有非数字类型；c. 除以 0）。同时编写整除函数 div，排除对应异常。在主函数中，测试异常的抛出。

第 8 章　面向对象编程

8.1　面向对象编程的概念

在现代编程开发中，“面向对象编程”是经常被提及的概念。那么，什么是面向对象编程呢？

在了解面向对象编程之前，我们先回头看看我们之前的编程方式。在我们之前编写的程序中，首先将问题分解成一步一步的操作，再按照操作的流程编写代码。这种编程思想被称为“面向过程编程”。面向过程编程的思想是最为直接的，它的好处在于思路更加清晰、更加符合程序执行的顺序。然而，在程序趋于复杂时，很多问题就会显现出来：

- 不易复用：当一个功能或对象需要在多个位置重复使用时，使用面向过程的编程方式需要将对应的代码复制多次，不易于编辑与修改。
- 不易拓展：当我们需要为之前的程序添加新功能时，使用面向过程的编程方式就会显得复杂，在拓展新功能时，很容易出现命名冲突、内存管理不当等问题。
- 不易维护：当被复制多次的代码需要修改时，显然，准确地修改多处将大大增加工作量并更容易出现人为的 bug。

在使用面向过程编程时，为了解决这些问题，我们将重复的代码封装成函数，一定程度地避免了以上一些问题，但是当程序逐渐复杂时，函数式的开发也很难满足开发者的需求。因此，“面向对象编程”的思想被提出来解决以上的问题。

“面向对象”（Object Oriented，OO）思想与以往的“面向过程”有很大区别，在“面向对象”的思想中，我们更多关心的是对象所具有的属性与方法，而不是事件的过程。以开车为例，按照以往“面向过程”的编程方法，如果想要发动车辆，需要按照以下的流程思考：

（1）点火；

（2）踩离合；

（3）踩油门；

（4）松离合。

而“面向对象”的思想，考虑的不是如何发动车辆，而是车上具有哪些可操作的零件来帮助发动，如：

- 汽车中有钥匙孔用来点火；
- 汽车中有离合器用来控制齿轮；
- 汽车中有油门用来控制发动机。

当提供了这些“零件”，使用者可以根据它们的使用方法来发动车辆。同时，当我们以汽

车为中心完成了设计时,还可以使用这个设计好的“模板”来制造很多同类的车,它们可能只有少数的属性和功能有差异,但大致上都符合汽车这类物体。从编程的角度来看,我们实际上是在对汽车这个类型进行封装与复用。

可见,面向对象的思想,更加符合自然中我们思考设计一类产品时的思想:它有哪些属性、它有哪些功能(方法)。由此可见,“属性”和“方法”是一个类最基本的特征,而根据这个类,我们可以创建出很多的对象,根据一个类创建的对象,我们称之为这个类的“实例”。

前文我们提过,“属性”和“方法”从编程层面实际上是对类型的“封装”,“封装”也是“面向对象编程”三大特性之一。除了“封装”,面向对象的特性还有“继承”与“多态”。在接下来的章节中,我们将围绕面向对象编程的这三个特性进行讲解。

8.2 类与对象

8.2.1 类与实例化

在上一节的例子中,我们设计了“汽车”这类物体,根据“汽车”这个类型,我们可以制造出很多汽车。实际上,这段话我们探讨的就是“类”与“对象”的关系。也就是说,“类”是对象的一种模板。例如,在 Python 中 Number 是一种类型,我们的代码“a = 1”和“b = 2”中,变量 a、b 都是 Number 类型的对象,也就是 Number 的实例;但是二者的实例属性的值不同,在这里 a 的值是“1”,b 的值是“2”。

在面向对象编程中,类的编写方法是必须要掌握的。

在使用自定义的类之前,我们需要声明一个类。声明类在 Python 中需要使用 class 关键字:

```
class 类名:
    类的内容……
```

在类结构中,我们可以声明变量或者函数。类结构中的变量对应类的“属性”,类结构中的函数对应类的“方法”。我们在接下来的章节中将介绍几类属性与方法。在这里,我们先从类的几个特殊的方法讲起。

创建类的对象的过程又叫作类的实例化。当类实例化时与实例化结束后,Python 会调用类的两个特殊的函数“构造函数”与“初始化函数”(当没有手动声明构造函数与初始化函数时,Python 会为其添加一个空的构造函数与初始化函数),在初始化中,我们可以对创建的实例进行初始化。同样,当一个对象不再被使用时,Python 会调用该类的“析构函数”,析构函数中可以对该对象使用的资源进行释放。

8.2.2 初始化函数与析构函数

Python 中初始化与虚构函数有固定的函数名,初始化的函数名为“__init__”,析构函数的函数名为“__del__”。每个类最多只能有一个初始化函数、一个析构函数。构造函数与析构函数都是类的“实例方法”(实例方法的具体概念将在本节后面介绍),也就是说二者

的第一个参数都应该是一个指向实例本身的 self 参数。在初始化函数中，我们还可以为其指定参数来对实例初始化，而析构函数不能有额外的参数。我们通过下面的例子学习初始化函数与析构函数的使用：

```
class Car:
    def __ init __(self,number):
        print "My Number is %s" % number
    def __ del __(self):
        print "I am destroyed !"
```

在这段代码中，我们声明了 Car 类并为其添加了初始化函数与析构函数，并在初始化函数中接收了一个参数 number。在 Car 类被实例化时，它将输出一个字符串与 Number；在 Car 类的实例销毁时，它同样会输出另一个字符串。

上面的代码中我们只声明了类却没有使用它，下面我们来将 Car 类实例化。实例化类的方法很简单，我们只需要在类名后使用括号，并在括号中填入初始化时需要的参数即可。这里需要注意的是 self 参数，self 参数是不需要人为传入的，Python 会在调用实例方法时自动传入 self 参数。self 参数是调用这个实例方法的实例本身的引用，即哪个对象调用了带有 self 参数的方法，self 就会指代谁。下面我们创建一个 Car 类的对象 c，并传入它的车牌号“10001”：

```
class Car:
    def __ init __(self,number):
        print "My Number is %s" % number
    def __ del __(self):
        print "I am destroyed !"

c = Car("10001")
```

这段代码执行后，我们会得到如图 8.1 的输出。

可以看到，当 Car 类对象 c 被实例化后，初始化函数被调用，输出了“My Number is 10001”，当程序执行结束对象 c 被销毁时，析构函数输出了“I am destroyed !”。

```
My Number is 10001
I am destroyed !
```

图 8.1　实例化对象

“类”与“对象”是面向对象编程的基石。而类的“属性”和“方法”，是一个类最基本的特征。那么，我们接下来将探讨类的“属性”与“方法”。

8.2.3　类的属性

在 Python 中，类的属性可以分为两种，一种是“类属性”，另一种是“实例属性”。无论是“类属性”还是“实例属性”，描述的都是一个类的特征，它们的区别在于：“类属性”在该类及其所有的实例中是共享的；而“实例属性”在实例之间不共享，每个实例都拥有一个属于实例本身的“实例属性”。

我们还是以汽车为例，汽车的总数，就可以看作汽车这个类的“类属性”，它被汽车这个类共享。而汽车的牌照则是汽车的“实例属性”，因为即使在同类汽车中，每个汽车都需要有

自己的牌照,汽车之间牌照互不影响。

在理解“类属性”与“实例属性”后,我们来学习它们在 Python 中的表达方式。

“类属性”只需要在声明中初始化即可,“类属性”会在类被导入时初始化。当我们使用一个类的类属性时,无论在类内还是在类外,我们只需要用“类名.类属性名”即可(当然,由于 Python 的语言机制,我们也可以在类代码外部动态地为类的类属性初始化,但是我们不推荐这种做法,因为如果在另一处使用这个类属性而它并未被初始化时,会引起 Python 的异常)。另外,由于 Python 的垃圾回收机制,类属性在析构时通过“类名.类属性名”的方式访问会出现引用异常,在析构函数中,我们应该使用“self.__class__.类属性名”的方式来访问类属性。

“实例属性”需要通过类中的实例函数中的 self 参数访问和初始化,由于“实例属性”必须存在于一个类的实例中,我们无法在类外通过“类名.属性名”的方式直接访问,而是通过“对象名.实例属性名”进行访问。

我们来改写之前的 Car 类代码,来体会二者的区别:

```
class Car:

    count = 0

    def __init__(self,number):
        Car.count += 1
        self.number = number
        print "My Number is %s" % number

    def __del__(self):
        self.__class__.count -= 1
        print "I am destroyed! My number is %s " % self.number

print Car.count
c1 = Car("10001")
print Car.count
c2 = Car("10002")
print Car.count
```

在这段代码中,我们为 Car 类加入了一个类属性 count 并初始化为 0,我们在初始化函数与析构函数中对“Car.count”进行增减,当有一个 Car 类实例被构造或析构时,相应的函数将会被调用来调整 count 值。同时,我们还为 Car 类的实例添加了一个实例属性 number,在初始化函数的“self.number = number”一句中,我们使用初始化函数的参数 number 的值来为实例属性 number 初始化;我们还在析构时让其再次输出这个 number。

运行这段代码,我们将得到如图 8.2 所示的输出。

8.2.4 类的方法

“方法”是指在类中定义的函数。Python 中的“方法”可分为三种:“实例方法”“静态方法”与“类方法”。

```
0
My Number is 10001
1
My Number is 10002
2
I am destroyed ! My number is 10002
I am destroyed ! My number is 10001
```

图 8.2　类属性与实例属性

“实例方法”是类的实例所特有的方法，实例方法只能通过实例的引用进行调用，并且在实例方法中可以通过 self 参数直接访问调用该方法的实例本身。例如类的初始化函数和析构函数都是实例方法，可以通过第一个参数 self 访问调用该方法的实例。在 Python 中，如果不使用特定的修饰器对类的函数进行修饰，在类中声明的方法默认为实例方法。实例方法需要在所有的参数之前添加一个指向调用该方法本身的参数，一般我们使用 self 作为该参数的名字。

“静态方法”既可以通过类名进行调用，也可以通过实例的引用进行调用。在 Python 中，静态方法在声明时需要使用修饰器“@staticmethod”修饰，即在函数声明的上一行中添加修饰器；同时，“静态方法”中不需要传入 self 参数，因此我们无法直接获取调用静态方法的对象的引用。Python 中的静态方法常用于工具函数的封装。例如我们自定义计算工具类 cal，并定义静态方法 add 为自定义的加法函数：

```
class cal:
    @staticmethod
    def add(x,y):
        return x + y

print cal.add(1,2)
```

运行后，将输出“1”加“2”的结果：“3”。

“类方法”同样既可以通过类名进行调用，也可以通过实例的引用进行调用。但是类方法需要传入一个参数（常命名为“cls”）作为调用该方法的类的引用，从这点上，类方法更像是实例方法，只不过它关注的不是调用该方法的对象而是类。Python 中类方法需要使用修饰器“@classmethod”进行修饰。例如，我们编写如下测试代码：

```
class C:
    name = 'C'
    @classmethod
    def foo(cls,content):
        print '%s : %s' % (cls.name,content)

C.foo("Hello")
c = C()
c.foo('Bye')
```

在这段代码中，我们定义了类 C，它拥有一个类方法 foo。在类方法 foo 中，我们使用 cls 参数访问调用该方法的类的属性 name。我们分别通过类和实例调用了该方法。这段代码运行后输出如图 8.3 所示。

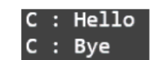


图 8.3　类方法测试

8.3　面向对象的三大特性

8.3.1　继承

在 8.2 节中，我们分别介绍了 Python 中的类的各种属性、方法的区别与使用。在设计类的属性与方法的时候，我们实际上是在对类进行“封装”操作，即类的开发者将类功能的细节写在类中，只留出必要的属性与方法让类的使用者使用，这样，使用者只需要知道开发者提供了哪些接口，而不需要了解其内部的具体实现，这个过程即为“封装”。例如我们的汽车，我们只需要知道转动方向盘可以转向，踩下油门可以加速，并不需要知道汽车中的每一个齿轮或电路的连接方式。在开发稍大的项目时，合适的封装显得尤为重要，因为封装可以提高复用性、方便维护与升级。

然而，有时我们需要比封装更加高级的功能来实现更复杂的类功能。例如，奔驰车和宝马车在一些属性和功能上有所不同，但是它们都是汽车，具有很多类似的功能与属性。如果单纯使用封装，我们需要把二者共有的属性和功能在各自的类中都重写一遍，这样对日后的维护很不友好，因此我们需要更高级的方式——“继承”。

“继承”是面向对象的第二个特性。与日常用语中的“继承”相似，如果一个“子类”继承于一个“父类”，“子类”即拥有父类的属性与方法，同时子类还可以拓展自己的属性或方法，因此，这句话中的“父类”也叫“基本类”或“基类”，“子类”也叫“衍生类”或“派生类”。

在 Python 中，我们需要在定义子类时声明继承关系。在声明类时，我们在类名后使用小括号，在小括号中填入该类需要继承的类型名即可(这个括号本质是一个元组，也就是说 Python 中一个类允许“多继承”，即一个子类继承多个父类)。例如，我们创建 Car 类并为其添加几个测试的属性与方法，再创建 BMW 类继承 Car 类：

```
class Car:
    def __init__(self,number):
        self.number = number
        print 'Car %s is constructed!' % number

    def horn(self):
        print 'Di,Di......'

    def move(self):
        print 'Car %s is moving!' % self.number

class BMW(Car):
    def __init__(self,number):
```

```
        print 'Car %s is a BMW !' % number

b = BMW('10001')
b.horn()
#b.move()
```

这段代码的运行效果如图 8.4 所示。

```
Car 10001 is a BMW !
Di,Di... ...
```

图 8.4 继承

在这个例子中，BMW 类继承了 Car 类，因此 BWM 类也具有 Car 类中的属性与方法，所以我们可以在 BMW 类的实例中使用 Car 类的实例方法 horn。

但是，如果我们取消最后一行的注释后，运行时将报错未定义属性 number，这是为什么呢？

在 Python 中，当子类没有声明初始化函数时，子类会自动执行父类的初始化函数；而当子类中声明了初始化函数时，子类不会自动调用父类的初始化函数，我们需要为其手动调用。因为子类的初始化函数与父类的初始化函数名相同，我们在调用父类同名的实例函数时，需要使用"类名.函数名"的方式调用，此时，我们需要手动传入 self 参数，而不能像"实例.函数名"的方式自动传入。我们对刚才的例子进行几处修改：

```
class Car:
    def __init__(self,number):
        self.number = number
        print 'Car %s is constructed !' % number

    def horn(self):
        print 'Di,Di... ...'

    def move(self):
        print 'Car %s is moving !' % self.number

class BMW(Car):
    def __init__(self,number):
        Car.__init__(self,number)
        print 'Car %s is a BMW !' % number

b = BMW('10001')
b.horn()
b.move()
```

修改后，BMW 的实例方法 move 可以正常访问 self.number 属性，如图 8.5 所示。

```
Car 10001 is constructed !
Car 10001 is a BMW !
Di,Di... ...
Car 10001 is moving !
```

图 8.5 手动调用父类初始化函数

子类除了可以直接调用父类的方法，还可以重写父类方法。重写父类方法时，我们只需要在子类中声明与父类具有同样函数名、参数列表的方法即可，例如在 BMW 中重写 Car 类的 move 方法：

```
class Car:
    def __init__(self,number):
        self.number = number
        print 'Car %s is constructed !' % number

    def horn(self):
        print 'Di,Di... ...'

    def move(self):
        print 'Car %s is moving !' % self.number

class BMW(Car):
    def __init__(self,number):
        Car.__init__(self,number)
        print 'Car %s is a BMW !' % number
    def move(self):
        print 'BWM %s is moving !' % self.number

b = BMW('10001')
b.horn()
b.move()
```

运行后，输出如图 8.6 所示，对象 b 执行了重写后的 move 方法：

```
Car 10001 is constructed !
Car 10001 is a BMW !
Di,Di... ...
BWM 10001 is moving !
```

图 8.6　方法重写

子类只有一个父类的继承叫作“单继承”，而子类具有多个父类的继承叫作“多继承”。Python 同样支持多继承方法。单继承与多继承本质上的区别不大，但是多继承在寻找方法或属性时更加复杂。

在之前的例子中，我们使用的都是单继承方法。在单继承中，当我们调用一个类的方法时，Python 会在该类中寻找是否有对应的方法，如果没有，则会在其父类中搜索，以此类推，直到找到对应方法，否则将抛出异常。然而在多继承中，寻找一个方法或属性的顺序就重要了起来。因为在一个类的多个父类中，可能出现同名的函数或方法，同时父类还可能存在自己的父类，在“父类的父类”中，也有可能存在同名方法或属性，因此，按照什么顺序搜索属性或方法成为多继承的一个问题。

在处理多继承问题的访问时，Python 2.x 中有如下规则：

- Python 2.x 中类根据其基类被分为两种：经典类与新式类。其中新式类需要继承 Python 中的 object 类型，而经典类不需要。
- 在搜索属性或方法时，新式类采用广度优先搜索，而经典类采用深度优先搜索。

我们以图 8.7 中的类为例：

图 8.7(a)的类 A、B、C、D 都直接或间接地由 object 类衍生，因此图 8.7(a)的类 A、B、C、D 均为新式类，图 8.7(b)的类 A、B、C、D 没有由 object 衍生，因此它们为经典类。

在新式类中，查找属性或方法时，按照广度优先查找，即当我们调用 D 类的对象中的 foo 方法时，Python 会按照 D→B→C→A→object 的顺序查找；在经典类中，按照深度优先查找，即 D→B→A→C 的顺序查找。当需要的方法或属性被找到时，查找将停止并调用找

到的方法或属性，如图 8.8 所示。

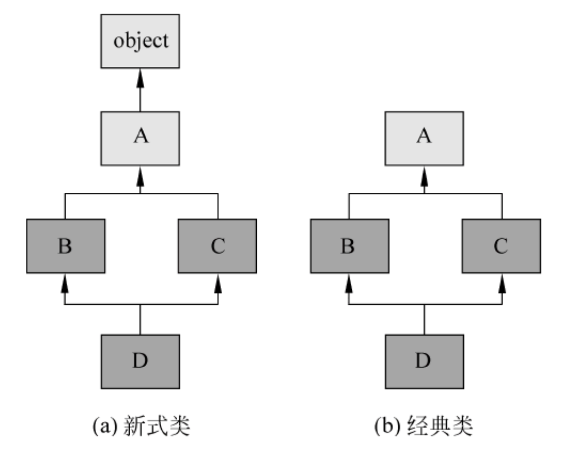


图 8.7　新式类与经典类

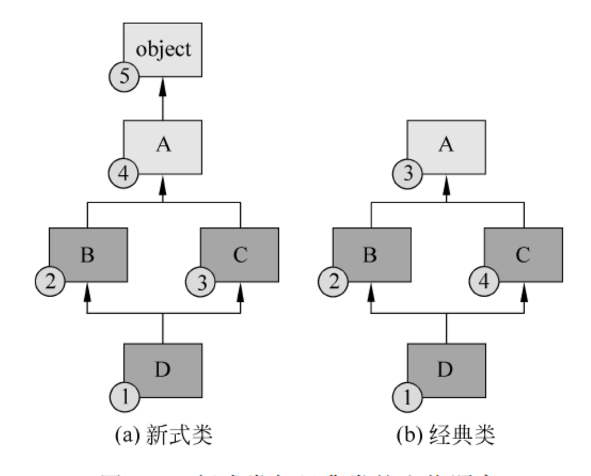


图 8.8　新式类与经典类的查找顺序

除了属性与方法的查找顺序，多继承中的初始化函数调用顺序同样是个问题。在单继承中，构造方法的调用规则比较简单：当子类没有声明初始化函数时，子类会自动执行父类的初始化函数；而当子类中声明了初始化函数时，子类不会自动调用父类的初始化函数，我们为其手动调用即可。而多继承中，调用规则如下：

- 如果子类中声明了初始化函数，则不会自动调用父类的初始化函数。
- 如果子类中没有声明初始化函数，则按照多继承查找方法的方式，调用找到的第一个初始化函数。

也就是说，多继承查找初始化函数时，同样是分别按照新式类与经典类查找一个方法的方式查找。我们通过如下的例子来验证：

```
class A(object):
    def __init__(self):
        print "A"

class B1(A):
```

```
        pass

class B2(A):
    def __init__(self):
        print "B2"

class C(B1,B2):
    pass

c = C()
```

这段代码是一个新式类多继承的例子，因为它们都直接或间接地由 object 类衍生而来，此时，初始化函数的查找按照广度优先搜索的顺序，因此运行后查找顺序与输出为“B2”。

接下来，我们修改类 A，使它不再继承于 object 类：

```
class A():
    def __init__(self):
        print "A"

class B1(A):
    pass

class B2(A):
    def __init__(self):
        print "B2"

class C(B1,B2):
    pass

c = C()
```

此时，这些类都是经典类，因为它们不直接或间接继承于 object。经典类的查找按照深度优先的顺序进行，因此，这段代码运行后将会输出“A”。

然而，从上面的例子可以看出，当子类没有声明初始化函数的时候，Python 只会找到一个初始化函数来完成构造，这时便又出现了之前的问题：如何让子类的每一个父类都完成构造？显然，我们还是可以手动调用父类的初始化函数，如下例所示：

```
class A(object):
    def __init__(self):
        print "enter A"
        print "leave A"

class B(A):
    def __init__(self):
        print "enter B"
```

```
        A.__init__(self)
        print "leave B"

class C(A):
    def __init__(self):
        print "enter C"
        A.__init__(self)
        print "leave C"

class D(B,C):
    def __init__(self):
        print "enter D"
        B.__init__(self)
        C.__init__(self)
        print "leave D"

d = D()
```

这段代码运行后，得到如下的输出：

```
enter D
enter B
enter A
leave A
leave B
enter C
enter A
leave A
leave C
leave D
```

可见，子类 D 的每个父类都完成了构造。然而，这里还存在一个问题。细心的读者可能发现，这里的 A 类被重复构造了两次。这时因为我们在类 B、C 中都调用了 A 类的初始化函数。这并不是我们希望得到的结果，我们希望子类的每个父类有且仅有一次构造。这时，需要使用 super 方法。

super 是 Python 2.x 中新式类特有的方法。使用“super.(父类名，自身引用)”可以访问其对应父类的方法。因此，在新式类中，我们也可以使用 super 的方式来代替“父类名.方法名”的方式来调用同名父类方法。在初始化函数中使用 super 来构造父类，可以保证父类只被构造一次：

```
class A(object):
    def __init__(self):
        print "enter A"
        print "leave A"
```

```
class B(A):
    def __init__(self):
        print "enter B"
        super(B,self).__init__()
        print "leave B"

class C(A):
    def __init__(self):
        print "enter C"
        super(C,self).__init__()
        print "leave C"

class D(B,C):
    def __init__(self):
        print "enter D"
        super(D,self).__init__()
        print "leave D"

d = D()
```

这段代码运行后输出如下：

```
enter D
enter B
enter C
enter A
leave A
leave C
leave B
leave D
```

可见，此时类 A 只被构造了一次。在多继承时，使用 super 构造父类是很重要的方法。而 super 实现的原理在于 Python 新式类中维护的 MRO 表，关于 MRO 表在这里不再详细讨论，感兴趣的读者可以自行查找资料了解 MRO 表的相关知识。

8.3.2 访问控制

在介绍类的属性和方法的一节中，我们把类与属性和方法关系看作“封装”。然而，这种“封装”仍然是“不完善”的封装。因为我们还是可以在类外访问到类的所有属性与方法的。然而，有时我们希望类的用户无法访问一些类的方法，那些属性与方法只能在类中或类的子类中被访问。就像汽车发动机的内部零件只有发明制造发动机的人能看到。这时，我们需要引入 Python 面向对象中的访问控制。

Python 的访问控制十分简单，它只有“公有”(public)与“私有”(private)的概念，而不像其他面向对象语言中还有“保护”(protected)属性。

Python 中的“公有”即无论在类中还是类外，我们都能访问该属性，如下例。

```
class A:
    def __init__(self):
        self.name = 'A'
    def foo(self):
        print self.name

a = A()

a.foo() # 在类中访问属性
print a.name # 在类外访问属性
```

在这个例子中,我们分别在类 A 中的 foo 函数与类 A 的实例 a 访问了类 A 的实例属性 name。由于类 A 的实例属性 name 是公有变量,因此这两种访问方式都是可行的。

而"私有"是指只能在类内部访问而不能在类外或类的子类中访问的属性。在 Python 中,想要声明私有变量,只需要将变量名以两个短下画线"__"开头即可。需要注意的是,私有变量只以两个下画线开头而不以两个下画线结尾,同时以两个下画线开头与结尾的是 Python 类的特殊属性和方法(如"__init__"):

```
class A:
    def __init__(self):
        self.__name = 'A'
    def foo(self):
        print self.__name

class B(A):
    def foo(self):
        print self.__name

a = A()
b = B()

a.foo()
#print a.__name          # 在类外访问私有变量,错误
#b.foo()                 # 在类的子类中访问私有变量,错误
#print b.__name          # 在子类的类外访问私有变量,错误
```

在这个例子中,我们为类 A 添加了私有变量"__name",它只有在类中如 foo 函数中可以访问,而在类外、子类中、子类外都无法直接访问,实现了对外的封装。需要注意的是,私有变量只是在类外或子类中无法访问,而其子类或实例仍含有这个变量。当然,这里的不能访问并不严谨,实际上,Python 解释器只不过为私有变量改了一下名字,我们仍可以通过改名后的变量名访问它,不过这里强烈推荐这样做,因为它破坏了封装。

8.3.3 多态

通俗来说,多态是指将一个子类对象当作其父类对象来使用,因为子类对象含有父类的所有方法与属性。例如我们之前进行异常处理时,就尝试过使用父类异常来代替子类来达

到捕获多种异常的目的。例如，我们编写如下例子：

```
class Animal(object):
    def speak(self):
        print self.words

class Cat(Animal):
    def __init__(self):
        self.words = 'mew mew'

class Dog(Animal):
    def __init__(self):
        self.words = 'wow wow'

def speak(animal):
    animal.speak()

cat = Cat()
dog = Dog()

speak(cat)
speak(dog)
```

在这个例子中，我们在 speak 函数中调用了 Animal 类的 speak 方法，而 Cat 和 Dog 都继承于 Animal 类，因此其都有 speak 方法，我们可以将 Cat 和 Dog 的实例当作父类类型使用。

在编译类面向对象语言中，多态是十分重要的，因为多态为其提供了很大的灵活性。而 Python 是动态的解释型语言，对象的属性和方法只有在使用时才会被检查。因此，在 Python 中没有严格意义上的多态，只要一个类拥有对应的属性和方法，都可以通过类似多态的方式来使用。例如，在上个例子中，即使一个与 Animal 类无关的类的实例也有 speak 方法，这样使用也不会报错。因此，可以说 Python 不具有严格的多态。

8.4 特殊的属性与方法

除了我们自定义的属性和方法外，Python 的类还有一些预设的特殊属性和特殊方法。这些属性或方法命名都以两个下画线起始与终止，如初始化函数“__init__”与析构函数“__del__”。这些属性和方法为它们操作类以及它们的对象提供了很多的便利。本节中，我们将分别讲解常用的特殊属性和方法的使用。

Python 中类的常用特殊属性与方法见表 8.1。

表 8.1 Python 常用的特殊属性与方法

名　称	描　述
可修改的特殊属性	
__slots__	一个包含字符串的元组，用来限制类允许添加的属性和方法名（只有在新式类中才能使用）

续表

名　　称	描　　述
只读的特殊属性	
__doc__	类的文档
__name__	类名
__module__	类所在的模块名
__bases__	类的基类，以元组返回
__dict__	类的所有成员，以字典类型返回
特殊方法	
__init__	初始化函数
__del__	析构函数
__str__	对象字符串化函数
__repr__	对象字符串化函数（命令行）

8.4.1　__slots__属性

Python作为一种动态的解释型语言，支持为类或对象动态添加属性或方法。如：

```
class Student(object):
    pass

s = Student()
s.name = 'Tom'
s.score = 100
```

在类的声明中，我们没为其添加任何属性或方法，而是为Student类的实例s添加了name与score属性。这样添加是被允许的。

而有时，我们想要限制允许添加的属性或方法，那么我们可以使用新式类的“__slots__”属性进行限制。“__slots__”属性是一个保存字符串的元组，使用“__slots__”属性后，我们只能为类的实例添加“__slots__”中包含的属性或方法。我们对之前的例子稍做修改：

```
class Student(object):
    __slots__ = ('name','age')

s = Student()
s.name = 'Tom'
s.score = 100
```

此时，在执行到“s.score = 100”时Python会报错Student类对象没有score属性。

8.4.2　只读的特殊属性

除了可以设置的“__slots__”属性外，Python的类还有很实用的只读特殊属性。这些属性可以帮助开发者查看类的信息。在表8.1中，我们已经列出了这些只读的属性，在这

里，我们通过代码实例实际验证一下这些特殊属性的用途：

```
class Student(object):
    '''This is a sample doc.'''
    name = None
    pass

print Student.__doc__
print Student.__name__
print Student.__module__
print Student.__bases__
print Student.__dict__
```

这段代码运行后的输出如图 8.9 所示。

```
This is a sample doc.
Student
__main__
(<type 'object'>,)
{'__dict__': <attribute '__dict__' of 'Student' objects>, '__module__':
'__main__', '__weakref__': <attribute '__weakref__' of 'Student' objects>,
'__doc__': 'This is a sample doc.', 'name': None}
```

图 8.9　只读的特殊属性

输出的内容分别为类的文档、类名、类所在的模块名、类的基类、类的成员字典。

8.4.3　__str__()方法

在编程中，我们经常需要输出一个对象的信息或将其转化为字符串进行处理。“__str__()”方法为写操作提供了很大的便利。

“__str__()”方法返回一个字符串。对于有“__str__()”方法的类的实例，我们可以使用“str(实例名)”将其转为字符串或直接使用“print 实例名”将其输出：

```
class Student(object):
    '''This is a sample doc.'''
    def __init__(self,id):
        self.id = id
    def __str__(self):
        return '< Student id = %s >' % self.id

s = Student(10001)
print str(s)
print s
```

代码运行后会得到如下输出：

```
< Student id = 10001 >
< Student id = 10001 >
```

8.4.4 __repr__()方法

“__repr__()”方法与“__str__()”方法类似，同样需要返回一个字符串作为该类对象字符串化的结果。不同的是，“__repr__()”可以用于命令行交互时直接输出对象。我们使用上一节的 Student 类，其中只用“__str__()”方法而没有“__repr__()”方法，进行命令行交互测试时输出如图 8.10 所示。

```
>>> from student import Student
>>> s = Student(10001)
>>> s
<student.Student object at 0x00000000029F3358>
```

图 8.10　没有__repr__()方法时的输出

可见，在命令行交互时，直接输出类时不会使用“__str__()”方法。

接着，我们为其添加“__repr__()”方法，为了简单，我们直接在“__repr__()”中返回“str(self)”：

```
class Student(object):
    '''This is a sample doc.'''
    def __init__(self,id):
        self.id = id
    def __str__(self):
        return '< Student id = %s >' % self.id
    def __repr__(self):
        return str(self)
```

再在命令行中进行输出，如图 8.11 所示。可见命令行输出调用了“__repr__()”方法。

```
>>> from student import Student
>>> s = Student(10001)
>>> s
< Student id=10001 >
```

图 8.11　使用__repr__()时的输出

习　题　8

一、选择题

1. 面向对象的三大特征中不包括(　　)。

　A. 继承　　B. 多态　　C. 对象　　D. 封装

2. 下面代码中，类(　　)不是新式类。

```
def A(object):
  pass
def B(A):
  pass
```

```
def C:
  pass
def D(A,B):
  pass
```

A. A　　B. B　　C. C　　D. D

3. 私有属性或方法名以(　　)开头。

A. #　　B. $　　C. __　　D. *

二、填空题

1. 面向对象的三大特性是________、________、________。

2. 从类创建对象的过程叫作类的________。类实例化的时候，会调用类的________；当对象被销毁时，会调用该对象的________。

3. 将属性与方法“打包”到类中，体现了面向对象三大特性的________。

4. 直接或间接由 object 类派生的类叫作________；不直接或间接继承于 object 类的类叫作________。

三、论述题

1. 简述类属性、实例属性的异同以及声明方式。

2. 简述实例方法、静态方法、类方法的异同以及生命方式。

3. 简述直接调用父类初始化函数与使用 super 调用初始化函数的异同。

四、编程题

编写学生类 Student，学生类有属性：学生姓名、学生生日、学生学号，这三个属性均为私有属性；学生类有方法：设置姓名、设置生日、设置学号、获取姓名、获取生日、获取学号，这些方法为公有方法(通过公有方法访问私有属性，这类函数被称为 getter 与 setter)。其中，学生生日类 Date 同样需要自定义类。Date 类需要三个属性，分别为年、月、日。

第9章 正则表达式

字符串是编程时不可忽视的数据结构,对字符串的操作有很多,有字符串的比较、连接等,其中字符串的匹配是一类典型操作。例如判断一个字符串是否符合要求的 IP 地址,或者是否为标准的手机号码,单纯靠编程逐个字符判断不仅困难还可能不准确。此时,正则表达式的存在就很好地解决了这类问题。

正则表达式又称规则表达式,通常被用来检索、替换那些符合某个模式的文本。使用正则表达式,可以快速地从一段文本中提取出我们想要的部分或对一段不需要的文本进行替换。

正则表达式不是 Python 的专利,不过 Python 为开发者提供了完善的正则表达式引擎以及对应的处理模块 re。本章我们将重点讲解正则表达式以及 re 模块的使用方法。

9.1 正则表达式模式

9.1.1 特殊字符

正则表达式可以按照规则匹配字符串,匹配的规则,我们同样使用一个字符串来表示,这个字符串被称为“模式串”。正则表达式引擎可以按照模式串来匹配字符串,由于需要匹配的字符串往往有一些部分是可变的,如我们需要匹配手机号以“139”开头、“0000”结尾的手机号。此时,我们需要一些特殊字符来表示这些可变的字符以及其他特殊字符等。正则表达式的特殊字符见表 9.1。

表 9.1 正则表达式常用特殊字符

<table>
<tr><th>特殊字符</th><th>描述</th></tr>
<tr><td>^</td><td>匹配字符串开头</td></tr>
<tr><td>$</td><td>匹配字符串末尾</td></tr>
<tr><td>.</td><td>匹配任意字符(注:不包含换行符,除非指定 re.DOTALL 属性)</td></tr>
<tr><td>\</td><td>转义字符,用来转义特殊字符使其被当作普通字符(如\$表示匹配字符$)</td></tr>
<tr><td>|</td><td>或(如 a|b 匹配 a 或者 b)</td></tr>
<tr><td>*</td><td>匹配 0 个或多个字符,如“1*”可以用来匹配“1”“11”“11111”</td></tr>
<tr><td>+</td><td>匹配至少 1 个字符,如“1+”可以匹配“11”“11111”</td></tr>
<tr><td>?</td><td>匹配 0 或 1 个字符,如“12?”可以匹配“1”或“12”</td></tr>
<tr><td>{n}</td><td>匹配 n 个字符,如“a(2)”可以匹配“aa”</td></tr>
<tr><td>{n,m}</td><td>匹配 n～m 个字符,如“a{1-3}”可以匹配“a”“aa”“aaa”</td></tr>
</table>

续表

特殊字符	描　述
[...]	匹配[]中任意一个字符,可以使用"-"表示连续的字符(如:[abc]则匹配'a'或'b'或'c',也可以写作[a-c])
[^...]	匹配除去[]中字符的字符(如[abc]匹配除去'a''b''c'之外的任一字符)
\w	匹配字母、数字及下画线
\W	匹配除字母、数字及下画线之外的任一字符
\s	匹配任意空白字符(相当于[\t\n\r\f])
\S	匹配任意非空字符(除\t\n\r\f之外的任一字符)
\d	匹配任意数字(相当于匹配[0-9])
\D	匹配任一非数字的字符
\n	匹配一个换行符
\t	匹配一个制表符

9.1.2　普通字符

特殊字符对应特殊的含义,相对地,我们还需要普通字符,普通字符只代表其本身。简单来说,符合以下规则的都是普通字符。

- 数字:0～9。
- 字母:a～z,A～Z。
- 其他非上述特殊字符的字符。

例如,我们需要匹配以"139"开头、"0000"结尾的手机号,可以使用如下的模式串:

```
139[0-9]{4}0000
```

9.1.3　特殊构造

有时,仅有普通字符与特殊字符仍满足不了我们的需求。例如,我们只需要匹配以"139"开头、"0000"结尾的手机号的中间4位,正则表达式提供了一些特殊构造来满足需求,这些特殊构造见表9.2。

表9.2　正则表达式的常用特殊构造

特殊构造	描　述
(...)	分组,括号中的内容作为一个组,每组作为一个整体。组具有编号,编号按照模式串从左往右遇到的左括号"("递增的方式确定。分组之后可加数量词。如"(abc){3}"可以匹配"abcabcabc"
(?P<别名>...)	分组,除了原有的编号外还可以指定别名
(?=...)	之后的字符串要匹配括号中的内容,但是匹配出的结果不包含括号中的内容。如"a(?=\d)"可以匹配"a1""a2"中的"a",而不能匹配"aa""ab"中的"a"
(?!...)	之后的字符串要不匹配括号中的内容,但是匹配出的结果不包含括号中的内容。如"a(?!\d)"可以匹配"aa""ab"中的"a",不能匹配"a1""a2"中的"a"

特殊构造	描　述
(?<=…)	之前的字符串要匹配括号中的内容，但是匹配出的结果不包含括号中的内容。如“(?<=\d)a”可以匹配“1a”“2a”中的“a”，而不能匹配“aa”“ba”中的“a”
(?<!…)	之前的字符串要不匹配括号中的内容，但是匹配出的结果不包含括号中的内容。如“(?<! \d)a”可以匹配“aa”“ba”中的“a”，而不能匹配“1a”“2a”中的“a”

正则表达式可以灵活地匹配需要的内容。想要掌握正则表达式需要大量的练习，读者可以在一些在线正则表达式测试的网站上练习匹配内容。常用的网站有：

- 开源中国正则表达式测试：http://tool.oschina.net/regex/。
- 站长工具正则表达式测试：http://tool.chinaz.com/regex/。

9.2　re 模块

掌握了正则表达式的模式串编写规则后，我们来学习使用 Python 的 re 模块来使用正则表达式进行字符串操作。

9.2.1　匹配模式

为了更加方便地处理复杂的文本，re 模块为模式串的匹配提供了一些模式，如表 9.3 所示。

表 9.3　re 模块匹配模式

模式标识	描　述
re.I	忽略大小写
re.M	多行模式，在多行模式下，“^”与“$”分别匹配每一行的开头和结尾
re.S	点模式，在点模式下，“.”可以匹配换行符
re.L	使预定字符类\w \W \b \B \s \S 根据当前设定的字符集匹配
re.U	使预定字符类\w \W \b \B \s \S 根据 Unicode 字符集匹配
re.X	详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释

如果需要使用这些模式，我们需要使用 re.compile 方法，将模式串与匹配规则编译 pattern 对象，re.comiple 函数的参数如下：

```
re.compile( 模式串 [ , 匹配模式标识] )
```

其中，匹配模式标识是可选的。如果需要使用多种模式，我们可以使用位运算中的或运算“|”。例如需要忽略大小写并使用点模式，此时，我们的标识参数应该填写“re.I|re.S”。

re 模块的模式串使用字符串类型作为参数，但是由于 Python 本身字符串也使用反斜线“\”进行转义，为了避免转义方式的混淆，我们一般使用“r”作为前缀来修饰模式串，如匹配“139”开头“0000”结尾的手机号时，我们的 Pattern 对象应如下得到：

```
import re
s = r"139\d{4}0000"
p = re.compile(s)
```

9.2.2 Pattern 对象

Pattern 对象是编译好的正则表达式，它不能直接实例化，而需要使用 re.compile 方法得到。Pattern 对象提供了一系列属性来获取相关信息与一系列方法对文本进行匹配。

Pattern 对象提供的属性如表 9.4 所示。

表 9.4 Pattern 对象的属性

属　性	描　述
pattern	编译时使用的正则表达式字符串
flags	编译时使用的匹配模式
groups	表达式中的分组数量
groupindex	以表达式中有别名的组的别名为键、以该组对应的编号为值的字典，没有别名的组不包含在内

我们通过下例测试这些属性：

```
import re

s = r'(139)(\d){4}(?P<tail>0000)'
p = re.compile(s,re.I)

print p.pattern
print p.flags
print p.groups
print p.groupindex
```

这段代码的输出如下：

```
(139)(\d){4}(?P<tail>0000)
2
3
{'tail': 3}
```

其中，flags 以数字的形式返回，我们可以通过位运算来验证其与我们使用的标识一致：

```
print p.flags ^ re.I
```

我们使用异或验算，输出为“0”，说明二者相等。

Pattern 对象提供了多种方法进行文本操作，同时 re 模块本身也提供了一些方法进行操作，二者的区别在于直接使用 re 模块的方法，需要额外传入模式串 pattern 与匹配模式

flags；而 Pattern 对象已经编译好了模式串与匹配模式 flags，因此不需要再次传入。Pattern 对象是可复用的，在大量多次匹配中，使用 Pattern 可以一定程度提高效率，而直接使用 re 模块方法仅可以省略一行 re.compile 操作。

Pattern 对象与 re 模块提供的匹配方法如下：

1．match

```
Pattern.match(文本[, 起始位置[, 终止位置]])
re.match(模式串, 文本[, 匹配模式标识])
```

从起始位置（re.match 方法无法手动设定起始位置，默认从下标 0 开始）匹配文本中内容与模式串内容，如果匹配成功返回一个 Match 对象，如果没有满足的匹配则返回 None。这里的匹配不是完全匹配，即只要文本中有能够匹配模式串的部分即可，如果需要完全匹配，可以在模式串中使用“$”来限定结尾边界。

我们通过下例测试 match 方法：

```
import re

p1 = r'abc'
p2 = r'abc$'

s1 = 'abcx'
s2 = 'abc'

print re.match(p1,s1)
print re.match(p1,s2)
print re.match(p2,s1)
print re.match(p2,s2)
```

这段代码的输出如下：

```
<_sre.SRE_Match object at 0x000000000333A510>
<_sre.SRE_Match object at 0x000000000333A510>
None
<_sre.SRE_Match object at 0x000000000333A510>
```

可见，返回了 None 的为失败的匹配，而返回 Match 对象的为成功匹配（Match 对象的使用我们将在下一节讲解）。通过“$”限定后，可以实现完全匹配。

2．search

```
Pattern.search(文本[, 起始位置[, 终止位置]])
re.search(模式串,文本[, 匹配模式标识])
```

search 用来在文本中查找可以匹配成功的子串。与 match 方法不同，search 方法并非从起始位置匹配，因此只要文本中存在匹配模式串的部分，就能匹配成功。同样，search 方

法成功匹配将返回一个 Match 对象，否则将返回 None。例如：

```
import re

p = r'abc'

s1 = '123abc123'
s2 = '123ac123'

pattern = re.compile(p)

print pattern.search(s1)
print pattern.search(s2)
```

这段代码的输出如下：

```
<_sre.SRE_Match object at 0x0000000002BBA510>
None
```

3. split

```
Pattern.split(文本[, 最大分割次数])
re.split(模式串, 文本[, 最大的分割次数])
```

按照匹配的子串将文本进行分割，以列表的方式返回。split 方法可以指定最大分割次数，超过最大分割次数之外的部分不会分割。split 的使用见下例：

```
import re

p = r'\d'

s = 'a1b2c3d4e5'

pattern = re.compile(p)

print pattern.split(s)
```

这段代码的输出如下：

```
['a', 'b', 'c', 'd', 'e', '']
```

4. findall

```
Pattern.findall(文本[, 起始位置[, 终止位置]])
re.findall(模式串, 文本[, 匹配模式标识])
```

findall 方法会以列表的方式返回全部匹配到的子串，例如：

```
import re

p = r'\d'

s = 'a1b2c3d4e5'

pattern = re.compile(p)

print pattern.findall(s)
```

这段代码的输出如下：

```
['1', '2', '3', '4', '5']
```

5. finditer

```
Pattern.finditer(文本[, 起始位置[, 终止位置]])
re.finditer(模式串, 文本[, 匹配模式标识])
```

finditer 的作用与 findall 类似，finditer 会匹配所有符合模式串的子串，不同的是 finditer 会以 Match 对象的迭代器的方式返回：

```
import re

p = r'\d'

s = 'a1b2c3d4e5'

pattern = re.compile(p)

for m in pattern.finditer(s):
    print m
```

这段代码的输出如下：

```
<_sre.SRE_Match object at 0x000000000264A510>
<_sre.SRE_Match object at 0x0000000002B41440>
<_sre.SRE_Match object at 0x000000000264A510>
<_sre.SRE_Match object at 0x0000000002B41440>
<_sre.SRE_Match object at 0x000000000264A510>
```

6．sub

```
Pattern.sub(替换的内容, 文本[, 最大替换次数])
re.sub(模式串, 替换的内容, 文本[, 最大替换次数])
```

sub 的作用是将匹配模式串的子串的内容进行替换。sub 有可选参数最大替换次数，默认替换全部匹配的子串。sub 函数将会返回替换后的文本：

```
import re

p = r'\d'

s = 'a1b2c3d4e5'

pattern = re.compile(p)

print pattern.sub('0',s)
```

这段代码的输出如下：

```
a0b0c0d0e0
```

7．subn

```
Pattern.subn(替换的文本, 文本[, 最大替换次数])
re.sub(模式串, 替换的文本, 文本[, 最大替换次数])
```

subn 与 sub 的功能一致，不同的是 subn 返回一个元组，元组中包含替换后的文本与替换次数：

```
import re

p = r'\d'

s = 'a1b2c3d4e5'

pattern = re.compile(p)

print pattern.subn('0',s)
```

这段代码的输出如下：

```
('a0b0c0d0e0', 5)
```

9.2.3 Match 对象

在 Pattern 与 re 的很多方法中，匹配成功时会返回 Match 对象。Match 对象是一个包含了很多匹配信息的对象，通过 Match 对象，我们可以获取除了匹配到的子串之外更多的信息。

Match 对象一系列属性与方法来获取这些信息，Match 对象提供的属性见表 9.5。

表 9.5　Match 对象的属性

属　性	描　述
string	匹配时使用的文本
re	匹配时使用的 Pattern 对象
pos	搜索时起始位置，与 Pattern. match()等方法中起始位置参数一致
endpos	搜索时终止位置，与 Pattern. match()等方法中终止位置参数一致
lastindex	最后一次匹配到的分组的索引，即匹配到的分组个数。如果没有分组被匹配则返回 None
lastgroup	最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组则返回 None

前 4 个属性的意义很容易理解，我们通过一个简单的例子来验证：

```
import re

p = r'(139)\d{4}(0000)'

s = 'abc13912340000bbb'

pattern = re.compile(p)

match = pattern.search(s,2,15)

print match.string
print match.re
print match.pos
print match.endpos
```

这段代码的输出如下：

```
abc13912340000bbb
<_sre.SRE_Pattern object at 0x0000000002ACB730>
2
15
```

lastindex 属性与 lastgroup 属性稍微有些难理解。其中“最后一个匹配的分组”是指最后一个右括号“)”的分组。我们可以通过如下例子来验证：

```
print re.search(r'(1)(2)(3)','123').lastindex
print re.search(r'((1)(2)(3))','123').lastindex

print re.search(r'(?P<g1>1)(?P<g2>2)(?P<g3>3)','123').lastgroup
print re.search(r'(?P<g4>(?P<g1>1)(?P<g2>2)(?P<g3>3))','123').lastgroup
```

这段代码的输出如下：

```
3
1
g3
g4
```

可见，这里的 last 指的不是最后开始的分组，而是最后结束的分组。

除了属性，Match 对象还有一系列方法提供匹配信息：

1. group([group1, …])

获得一个或多个分组截获的字符串，指定多个参数时将以元组形式返回。其中，可选参数 group1 等可以使用分组的编号或别名，编号 0 代表整个匹配的子串；不填写参数时，返回 group(0)。如果没有截获字符串的组返回 None。截获了多次的组返回最后一次截获的子串。例如：

```
import re

p = r'(139)(\d{4})(0000)'

s = '13912340000'

pattern = re.compile(p)

print pattern.search(s).group(0,1,2,3)
```

这段代码的输出如下：

```
('13912340000', '139', '1234', '0000')
```

2. groups([default])

以元组形式返回全部分组截获的字符串。相当于调用 group(1,2,…,last)。default 表示没有截获字符串的组以这个值替代，默认为 None。例如：

```
import re

p = r'(139)(\d{4})(0000)'

s = '13912340000'
```

```
pattern = re.compile(p)
print pattern.search(s).groups()
```

这段代码的输出如下：

```
('139', '1234', '0000')
```

3. groupdict([default])

返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。如果为空，则以 default 参数代替，默认为 None。例如：

```
import re
p = r'(139)(?P<mid>\d{4})(0000)'
s = '13912340000'
pattern = re.compile(p)
print pattern.search(s).groupdict()
```

这段代码的输出如下：

```
{'mid': '1234'}
```

4. start([group])、end([group])、span([group])

start([group])、end([group])分别返回指定的组截获的子串在文本中的起始索引(子串的第一个字符索引)或终止索引(子串的最后一个字符索引+1)。group 默认值为 0。而 span([group])返回元组(start(group), end(group))。例如：

```
import re
p = r'(139)(?P<mid>\d{4})(0000)'
s = 'abcd13912340000abcd'
pattern = re.compile(p)
match =  pattern.search(s)
print match.start(),match.end(),match.span()
```

这段代码的输出如下：

```
4 15 (4, 15)
```

5．expand(template)

将匹配到的分组代入 template 中然后返回。template 相当于返回格式的模板，其中可以使用“\分组索引”“\g <分组索引>”或“\g <分组别名>”引用分组，但不能使用编号 0。“\分组索引”与“\g <分组索引>”是等价的；但“\10”将被认为是第 10 个分组，如果想表达“\1”之后是字符‘0’，只能使用“\g < 1 > 0”。例如：

```
import re
p = r'(139)(?P<mid>\d{4})(0000)'
s = 'abcd13912340000abcd'
pattern = re.compile(p)
print pattern.search(s).expand('\g<3> \g<mid> \g<1>')
```

这段代码的输出如下：

```
0000 1234 139
```

习　题　9

一、选择题

1．如果我们想匹配以“123”开头的子串，需要使用(　　)来限定开头。

A．%　　　B．^　　　C．&　　　D．$

2．如果想匹配以“123”开头的子串，但是我们想去掉“123”，应该使用(　　)结构。

A．(? <...)　　　B．(?!...)　　　C．(? <=...)　　　D．(?! =...)

3．re 模块或 Pattern 的(　　)方法用于从起始位置开始匹配。

A．search　　　B．findall　　　C．match　　　D．finditer

二、填空题

1．在 re 模块的匹配模式中，忽略大小写的标识符是________，多行模式的标识符是________，点模式的标识符是________，详细模式的标识符是________。

2．使用 Pattern 对象方法与直接使用 re 模块的函数，________需要先进行编译。

3．Match 对象的 group 方法用于返回匹配到的分组，若我们需要返回完整的匹配子串，group 的参数需要填________，或者默认不添加参数。

三、论述题

1．简述模式串中特殊字符、特殊结构以及它们的含义。

2．简述 Pattern 对象的匹配方法。

3. 简述 Match 对象的方法。

四、编程题

爬虫是目前互联网常用的技术，爬虫按照一定规则访问页面并保存 HTML 页面中有用的信息。在获取信息时，xpath 和正则表达式是常用的提取方法。试从如下 HTML 代码中，使用正则表达式匹配电影别名、发行年份、主演、导演、编剧等信息。

```
<div id = "info">
                <span ><span class = 'pl'>导演</span>: <span class = 'attrs'><a href =
"/celebrity/1047973/" rel = "v:directedBy">弗兰克·德拉邦特</a></span></span><br/>
                <span ><span class = 'pl'>编剧</span>: <span class = 'attrs'><a href =
"/celebrity/1047973/">弗兰克·德拉邦特</a> / <a href = "/celebrity/1049547/">斯蒂芬·金</a>
</span></span><br/>
                <span class = "actor"><span class = 'pl'>主演</span>: <span class = 'attrs'>
<a href = "/celebrity/1054521/" rel = "v:starring">蒂姆·罗宾斯</a> / <a
                href = "/celebrity/1054534/" rel = "v:starring">摩根·弗里曼</a> / <a
                href = "/celebrity/1041179/" rel = "v:starring">鲍勃·冈顿</a> / <a
                href = "/celebrity/1000095/" rel = "v:starring">威廉姆·赛德勒</a> / <a
                href = "/celebrity/1013817/" rel = "v:starring">克兰西·布朗</a> / <a
                href = "/celebrity/1010612/" rel = "v:starring">吉尔·贝罗斯</a> / <a
                href = "/celebrity/1054892/" rel = "v:starring">马克·罗斯顿</a> / <a
                href = "/celebrity/1027897/" rel = "v:starring">詹姆斯·惠特摩</a> / <a
                href = "/celebrity/1087302/" rel = "v:starring">杰弗里·德曼</a> / <a
                href = "/celebrity/1074035/" rel = "v:starring">拉里·布兰登伯格</a> / <a
                href = "/celebrity/1099030/" rel = "v:starring">尼尔·吉恩托利</a> / <a
                href = "/celebrity/1343305/" rel = "v:starring">布赖恩·利比</a> / <a
                href = "/celebrity/1048222/" rel = "v:starring">大卫·普罗瓦尔</a> / <a
                href = "/celebrity/1343306/" rel = "v:starring">约瑟夫·劳格诺</a> / <a
                href = "/celebrity/1315528/" rel = "v:starring">祖德·塞克利拉</a></span>
</span><br/>
                <span class = "pl">类型:</span><span property = "v:genre">剧情</span> /
<span property = "v:genre">犯罪</span><br/>

                <span class = "pl">制片国家/地区:</span>美国<br/>
                <span class = "pl">语言:</span>英语<br/>
                <span class = "pl">上映日期:</span><span property = "v:initialReleaseDate"
content = "1994 - 09 - 10(多伦多电影节)">1994 - 09 - 10(多伦多电影节)</span> / <span
property = "v:initialReleaseDate" content = "1994 - 10 - 14(美国)">1994 - 10 - 14(美国)</span>
<br/>
                <span class = "pl">片长:</span><span property = "v:runtime" content = "142"
>142 分钟</span><br/>
                <span class = "pl">又名:</span>月黑高飞(港) / 刺激 1995(台) / 地狱诺言 / 铁
窗岁月 / 肖申克的救赎<br/>
                <span class = "pl"> IMDb 链接:</span><a href = "http://www.imdb.com/title/
tt0111161" target = "_blank" rel = "nofollow">tt0111161 </a><br>
</div>
```

第 10 章　Python GUI 编程

10.1　GUI 编程简介

之前我们所学的程序都是控制台程序。除此之外，我们还可以使用 GUI 编程。

10.1.1　GUI 编程

GUI(Graphical User Interface，图形用户界面，又称图形用户接口)是指采用图形方式显示的计算机操作用户界面。与命令行界面相比，图形界面程序不需要用户死记硬背大量指令，更易于接受，也更加美观。学习 GUI 编程，可以让开发者写出更加简洁、易用的图形用户界面程序。

10.1.2　GUI 编程的特点

GUI 编程的优点：

- 易用，不需要用户死记硬背大量指令。
- 美观，图形用户界面程序在视觉上更容易被用户接受。
- 功能强大，GUI 程序往往可以提供更加丰富的功能。

GUI 编程的缺点：

- 复杂，相比控制台程序，GUI 程序代码量更多、更复杂。
- 资源消耗多，相比控制台程序，GUI 程序会占用更多资源。
- 稳定性差，若代码不够健壮，可能会出现 UI 线程被阻塞而停止响应的问题。

10.1.3　Python GUI 编程

Python 为 GUI 编程提供了很多开发工具，例如 Tkinter、Qt、wxPython 等。每种开发工具有各自的优缺点，适用于不同场合。

本章我们将介绍 Tkinter，Tkinter 是 Python 的标准 GUI 库。Python 使用 Tkinter 可以快速地创建 GUI 应用程序。它也是以上各种开发工具中最易于初学者使用的。同时，这些 GUI 知识也同样适用于大多数其他 GUI 框架。

10.2 Tkinter 模块 GUI 编程基础

10.2.1 Tkinter 基础

Tkinter GUI 主要由窗体、组件、布局组成。而组件与布局都依附于窗体。

使用 Tkinter 编程一般分为如下 4 个步骤：

(1) 导入 Tkinter 模块。

(2) 创建并初始化窗体。

(3) 创建组件与布局并为其绑定父窗体。

(4) 进入事件循环。

下面我们通过一个 GUI Hello World 程序来感受这个过程。

```
# -*- coding: utf-8 -*-

#①
#导入 Tkinter 模块(为了方便并清晰地看出哪些内容属于 Tkinter,此处将 Tkinter 模块重命名为
tk)
import Tkinter as tk

#②
#创建并初始化新窗体 root
root = tk.Tk()

#将 root 窗体的名称改名为 F
root.title('F')

#③
#创建 Label 类型组件 label_1,将其绑定在父对象 root 上并设置文本内容
label_1 = tk.Label(root, text = "Hello World !\n I love Python !\n Let's learn Python
together !")

#将 label_1 以 pack 的布局显示
label_1.pack()

#④
#进入事件循环
root.mainloop()
```

效果如图 10.1 所示。

如上代码中,①②③④分别对应上文提到的 4 个步骤。在这个例子中,我们可以看出,①②④步骤主要决定窗体,而 GUI 编程中,我们往往更关注③步骤中窗体内各种组件布局的实现。因此,简单的窗体程序中①②④部分基本相同,我们只需要对③进行修改,就能实现各种样式的 GUI。

下面将详细讲解③部分中组件与布局的实现。

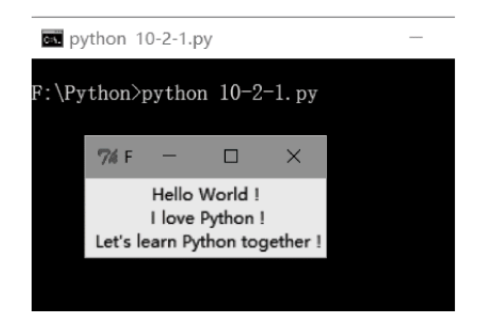


图 10.1　Tkinter 示例

```
label_1 = tk.Label(root,text = "Hello World !\n I love Python !\n Let's learn Python together !")
```

在这段代码中,我们创建了一个 Label 类型的变量 label_1。Label 是 Tkinter 各种组件之一,主要用来显示文本与图片。Tkinter 中有很多组件,表 10.1 展示的是 Tkinter 中常用的组件。

表 10.1　Tkinter 常用组件

组　件	功　能
Button	按钮组件,在程序中显示按钮
Canvas	画布组件,显示图像元素(如线条、文本等)
CheckButton	多选框组件,用于在程序中提供多项选择框
Entry	输入组件,用于显示简单的文本内容
Frame	框架组件,在屏幕上显示一个矩形区域,多用来作为容器
Label	标签组件,可以显示文本和位图
Listbox	列表框组件,在 Listbox 窗口中此部件用来显示一个字符串列表给用户
MenuButton	菜单按钮组件,用于显示菜单项
Menu	菜单组件,显示菜单栏,下拉菜单和弹出菜单
Message	消息组件,用来显示多行文本,与 label 比较类似
RadioButton	单选按钮组件,显示一个单选的按钮状态
Scale	范围组件,显示一个数值刻度,为输出限定范围的数字区间
Scrollbar	滚动条组件,当内容超过可视化区域时使用,如列表框
Text	文本组件,用于显示多行文本
Toplevel	容器组件,用来提供一个单独的对话框,与 Frame 比较类似
SpinBox	输入组件,与 Entry 类似,但是可以指定输入范围值
PanedWindow	窗口布局管理的组件,可以包含一个或者多个子控件
LabelFrame	容器组件,常用于复杂的窗口布局
tkMessageBox	消息框组件,用于显示消息

感受了 Tkinter 组件的丰富之后,我们再来看上一段代码。

在调用 Label 的初始化函数来初始化 label_1 时,我们传递了两个参数。其中第一个参

数是 label_1 的父级对象，也就是 label_1 依附于哪个对象存在，这一参数在对 GUI 布局管理时尤为重要，我们会在之后的例子中进行讲解；第二个参数是 Label 组件特有的属性，用来修改所显示的文本内容。

在 Tkinter 中，每个组件除了有自己特有的属性(如 Label 的 text 属性)外，还具有通用属性。通用属性是大部分 Tkinter 组件都具有的属性，包括大小、字体和颜色等，详见表 10.2。

表 10.2　Tkinter 通用属性

选项(别名)	说　明	单位	典型值	没有此属性的控件
background(bg)	当控件显示时，给出的正常颜色	color	'gray25' '#ff4400'	
borderwidth(bd)	设置一个非负值，该值显示画控件外围 3D 边界的宽度(特别的由 relief 选项决定这项决定)；控件内部的 3D 效果也可以使用该值，该值可以是 Tkinter(Tk_GetPixels)接受的任何格式	pixel	3	
cursor	指定控件使用的鼠标光标，该值可以是 Tkinter(Tk_GetPixels)接受的任何格式	cursor	gumby	
Font	指定控件内部文本的字体	font	'Helvetica' ('Verdana',8)	Canvas Frame Scrollbar Toplevel
foreground(fg)	指定控件的前景色	color	'black' '#ff2244'	Canvas Frame Scrollbar Toplevel
highlightbackground	指出经过没有输入焦点的控件加亮区域颜色	color	'gray30'	Menu
Highlyghtcolor	指出经过没有输入焦点的控件周围长方区域加亮颜色	color	'royalblue'	Menu
highlightthickness	设置一个非负值，该值指出一个有输入焦点的控件周围加亮方形区域的宽度，该值可以是 Tk_GetCursor)接受的任何格式。如果为 0，则不画加亮区域	pixel	2.1m	Menu
relief	指出控件 3D 效果。可选值为 RAISED，SUNKEN，FLAT，RIDGE，SOLID，GROOVE。该值指出控件内部相对于外部的外观样式，例如 RAISED 意味着控件内部相对于外部突出	constant	RAISED GROOVE	

续表

选项(别名)	说　　明	单位	典型值	没有此属性的控件
takefocus	决定窗口在键盘遍历时是否接收焦点(例如 Tab,Shift-Tab)。在设定焦点到一个窗口之前,遍历脚本检查 takefocus 选项的值,值 0 意味着键盘遍历时完全跳过,值 1 意味着只要有输入焦点(它及所有父代都映射过)就接收。空值由脚本自己决定是否接收,当前的算法是如果窗口被禁止,或者没有键盘捆绑或窗口不可见时,跳过	boolean	1 YES	
Width	指定一个整数,设置控件宽度,控件字体的平局字符数.如果值小于等于 0,控件选择一个能够容纳目前字符的宽度	integer	32	Menu

除了通用属性之外,还有很多属性被一系列控件共有,详见表 10.3。

表 10.3　Tkinter 组件共有属性

选项(别名)	说　　明	单位	典型值	仅此类控件
Activebackground	指定画活动元素的背景颜色。元素(控件或控件的一部分)在鼠标放在其上并按动鼠标按钮引起某些行为的发生时,是活动的。如果严格的 Modf 一致性请求通过设置 tk_strictModf 变量完成,该选项将被忽略,正常背景色将被使用。对 Windows 和 Macintosh 系统,活动颜色将只有在鼠标按钮 1 被按过元素时使用	color	'red' '#fa07a3'	Button Checkbutton Menu Menubutton Radiobutton Scale Scrollbar
activeforeground	指定画活动元素时的前景颜色。参见上面关于活动元素的定义	color	'cadeblue'	Button Menu Checkbutton Menubutton Radiobutton
anchor	指出控件信息(例如文本或者位图)如何在控件中显示。必须为下面值之一:N,NE,E,SE,S,SW,W,NW 或者 CENTER。例如 NW(NorthWest)指显示信息时使左上角在控件的左上端	constant		Button Checkbutton Label Message Menubutton Radiobutton

续表

选项(别名)	说　　明	单位	典型值	仅此类控件
bitmap	指定一个位图在控件中显示,以 Tkinter (Tk_GetBitmap)接受的任何形式。位图显示的精确方式受其他选项如锚或对齐的影响。典型的,如果该选项被指定,它覆盖指定显示控件中文本的其他选项;bitmap 选项可以重设为空串以使文本能够显示在控件上。在同时支持位图和图像的控件中,图像通常覆盖位图	bitmap		Button Checkbutton Label Menubutton Radiobutton
command	指定一个与控件关联的命令。该命令通常在鼠标离开控件时被调用,对于单选按钮和多选按钮,tkinter 变量(通过变量选项设置)将在命令调用时更新	command	setupData	Button Checkbutton Radiobutton Scale Scrollbar
disabledforeground	指定绘画元素时的前景色。如果选项为空串(单色显示器通常这样设置),禁止的元素用通常的前景色画,但是采用点刻法填充模糊化	color	'gray50'	Button Checkbutton Radiobutton Menu Menubutton
height	指定窗口的高度,采用字体选项中给定字体的字符高度为单位,至少为 1	integer	1　4	Button Canvas Frame Label Listbox Checkbutton Radiobutton Menubutton Text Toplevel
image	指定所在控件中显示的图像,必须是用图像 create 方法产生的。如果图像选项设定,它覆盖已经设置的位图或文本显示;更新恢复位图或文本的显示需要设置图像选项为空串	image		Button Checkbutton Label Menubutton Radiobutton
justify	当控件中显示多行文本的时候,该选项设置不同行之间是如何排列的,其值为如下之一: LEFT,CENTER 或 RIGHT。LEFT 指每行向左对齐,CENTER 指每行居中对齐,RIGHT 指向右对齐	constant	RIGHT	Button Checkbutton Entry Label Menubutton Message Radiobutton

续表

选项(别名)	说　　明	单位	典型值	仅此类控件
padx	指定一个非负值设置控件 X 方向需要的边距。该值为 Tkinter(Tk_GetPixels)接受的格式。当计算需要多大的窗口时,控件会把此值加到正常大小之上(由控件中显示内容决定);如果几何管理器能够满足此请求,控件将在左端或右端得到一个给定的多余空边。大部分控件只用此项于文本,如果它们显示位图或图像,通常忽略空边选项	pixels	2m　10	Button Checkbutton Label Menubutton Message Radiobutton Text
pady	指定一个非负值设置控件 Y 方向需要的边距。该值为 Tkinter(Tk_GetPixels)接受的格式。当计算需要多大的窗口时,控件会把此值加到正常大小之上(由控件中显示的内容决定);如果几何管理器能够满足此请求,控件将在上端或下端得到一个给定的多余空边。大部分控件只用此项于文本,如果它们显示位图或图像,通常忽略空边选项	pixels	12　3m	Button Checkbutton Label Menubutton Message Radiobutton Text
selectbackground	指定显示选中项时的背景颜色	color	blue	Canvas Listbox Entry Text
selectborderwidth	指定一个非负值,给出选中项的三维边界宽度,值可以是任何 Tkinter(Tk_GetPixels)接受的格式	pixel	3	Canvas Entry Listbox Text
selectforeground	指定显示选中项的前景颜色	color	yellow	Canvas Entry Listbox Text
state	指定控件下列两三个状态之一(典型是复选按钮): NORMAL 和 DISABLED 或 NORMAL,ACTIVE 和 NORMAL。在 NORMAL 状态,控件有前景色和背景显示;在 ACTIVE 状态,控件按 activeforeground 和 activebackground 选项显示;在 DISABLED 状态下,控件不敏感,缺省捆绑将拒绝激活控件,并忽略鼠标行为,此时,由 disabled foreground 和 background 选项决定如何显示	constant	ACTIVE	Button Checkbutton Entry Menubutton Scale Radiobutton Text

续表

选项(别名)	说　　明	单位	典型值	仅此类控件
text	指定控件中显示的文本,文本显示格式由特定控件和其他诸如锚和对齐选项决定	string	'Display'	Button Checkbutton Label Menubutton Message Radiobutton
textvariable	指定一个变量名字。变量值被转变为字符串在控件上显示。如果变量值改变,控件将自动更新以反映新值,字符串显示格式由特定控件和其他诸如锚和对齐选项决定	variable	widgetConstant	Button Checkbutton Entry Label Menubutton Message Radiobutton
underline	指定控件中加入下画线字符的整数索引。此选项完成菜单按钮与菜单输入的键盘遍历默认捆绑。0 对应控件中显示的第一个字符,1 对应第二个,以此类推	integer	2	Button CheckButton Label Menubutton Radiobutton
wraplength	对于能够支持字符换行的控件,该选项指定行的最大字符数,超过最大字符数的行将转到下行显示,这样一行不会超过最大字符数。该值可以是窗口距离的任何标准格式。如果该值小于或等于 0,不换行,换行只在文本中换行符的地方才出现	pixel	41,65	Button Checkbutton Label Menubutton Radiobutton
xscrollcommand	指定一个用来与水平滚动框进行信息交流的命令前缀,当控件窗口视图改变(或者别的任何滚动条显示的改变,如控件的总尺寸改变等),控件将通过把滚动命令和两个数连接起来产生一个命令。两个数分别为 0 到 1 之间的分数,代表文档中的一个位置,0 表示文档的开头,1.0 表示文档的结尾处,0.333 表示整个文档的 1/3 处,如此等等。第一个分数代表窗口中第一个可见文档信息,第二个分数代表紧跟上一个可见部分之后的信息。然后命令把它们传到 TCL 解释器执行。 典型的,xscrollcommand 选项由滚动条标识跟着 set 组成,如 set. x. scrollbar set 将引起滚动条在窗口中视图变化时被更新。如果此项没有指定,不执行命令	function		Canvas Entry Listbox Text

续表

选项(别名)	说　明	单位	典型值	仅此类控件
yscrollcommand	指定一个用来与垂直滚动框进行信息交流的命令前缀，当控件窗口视图改变(或者别的任何滚动条显示的改变，如控件的总尺寸改变等)，控件将通过把滚动命令和两个数连接起来产生一个命令。两个数分别为 0 到 1 之间的分数，代表文档中的一个位置，0 表示文档的开头，1.0 表示文档的结尾处，0.333 表示整个文档的 1/3 处，如此等等。第一个分数代表窗口中第一个可见文档信息，第二个分数代表紧跟上一个可见部分之后的信息。然后命令把它们传到 TCL 解释器执行。 典型的，yscrollcommand 选项由滚动条标识跟着 set 组成，如 set.y.scrollbar set 将引起滚动条在窗口中视图变化时被更新。如果此项没有指定，不执行命令	function		Canvas Entry Listbox Text

在理解这一行代码之后，我们来看下一行代码：

```
label_1.pack()
```

这段代码，我们使用 tk 组件的 pack()方法对 label_1 进行布局。pack()是 Tkinter 提供的布局方法之一，它的功能是根据内容自动调整大小，因此，在上个例子中，我们的窗体创建时便会刚好容纳 label_1 内的文本内容。

Tkinter 为我们提供了三种常用的布局方法，详见表 10.4。

表 10.4　Tkinter 布局方法

几 何 方 法	描　述
grid()	网格，网格布局
pack()	包装，自动调整
place()	位置，根据位置布局

在后面的例子中，我们会详细介绍这几种布局。

10.2.2　Tkinter 组件

在 10.2.1 节中，我们学习了 Tkinter 程序的基本结构，本节我们将分别讲解 Tkinter 各种常用组件的用法。

1. Label 标签

Label 是用来显示文本、图片等内容的组件。Label 显示文本时，可以使用 text 属性显示文本常量，也可以使用 textvariable 显示 StringVar 类型的文本变量。Label 既可以单独显示文本或图片，也可以同时显示文本与图片。在同时显示文本与图片时，需要修改 Label

对象的 compound 属性，compound 支持 text、image、top、center、left、right 等图文混排方式。下面我们通过一个例子实践以上内容：

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()

# 可以在创建 label_1 时传递静态文本
label_1 = tk.Label(root, text = "You cannot see me !")
label_1["text"] = "I'm label_1"

# 可以将 Label 对象的 textvariable 属性设置为一个 StringVar 对象,动态修改文本
label_2 = tk.Label(root)
txt = tk.StringVar()
label_2['textvariable'] = txt
txt.set("You cannot see me !")
txt.set("I'm label_2")

# 可以显示 gif 图片
label_3 = tk.Label(root)
img_3 = tk.PhotoImage(file = "res/img_pylogo.gif")
label_3["image"] = img_3

# 可以同时显示图片,但是要修改 Label 对象的 compound 属性
label_4 = tk.Label(root)
img_4 = tk.PhotoImage(file = "res/img_pylogo.gif")
label_4["image"] = img_4
label_4["text"] = "I'm label_4"
label_4["compound"] = "right"

label_1.pack()
label_2.pack()
label_3.pack()
label_4.pack()

root.mainloop()
```

这段代码的运行效果如图 10.2 所示。

2. Button 按钮

Button 是 Tkinter 提供的按钮类，与 Label 类似，Button 也支持文本按钮、图片按钮、图文混排按钮。使用图文按钮时，也需要修改 compound 属性。Button 有一个重要的属性 command，command 可以赋值为一个函数对象，在单击按钮时，command 所指的函数便会执行。

我们通过如下例子，学习这些按钮的使用方法。

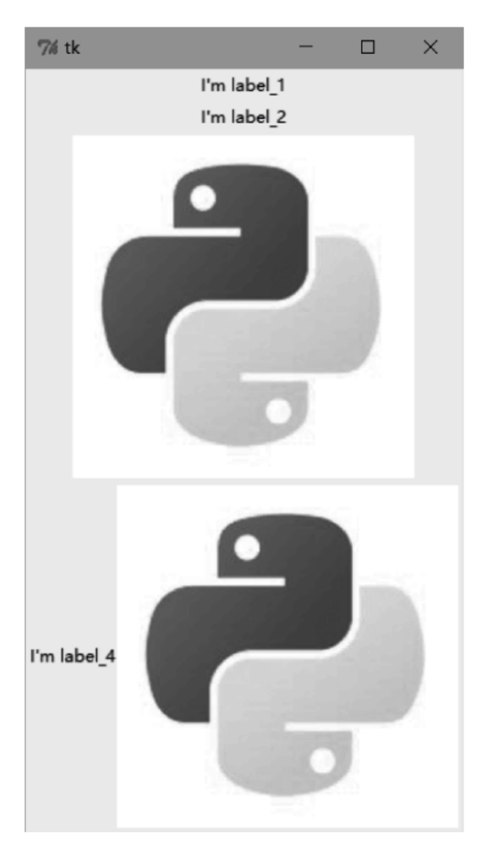


图 10.2　Label 组件

```
# -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

label = tk.Label(root, text="Please press the buttons !")

# 简单的按钮,单击后可以修改 Label 内的文本

def func_1():
    label["text"] = "You've clicked button_1 !"

button_1 = tk.Button(root, text="Button_1", command=func_1)

# 图片按钮,与 Label 显示图片类似

def func_2():
```

```
    label["text"] = "You've clicked button_2 !"

image_2 = tk.PhotoImage(file="res/img_button.gif")
button_2 = tk.Button(root, image=image_2, command=func_2)

# Button 对象也支持图文混排,与 Label 对象类似,同样需要修改 compound 属性即可

def func_3():
    label["text"] = "You've clicked button_3 !"

image_3 = tk.PhotoImage(file="res/img_button.gif")
button_3 = tk.Button(root,text='Button_3',image=image_3,compound='right',command=func_3)

# 使用 pack 包装

label.pack()
button_1.pack()
button_2.pack()
button_3.pack()

# Button 提供了一些效果:flat, groove, raised, ridge, solid, sunken
# 以下按钮只用于展示效果,没有绑定指令

tk.Button(root, text='FLAT', relief=tk.FLAT).pack()
tk.Button(root, text='GROOVE', relief=tk.GROOVE).pack()
tk.Button(root, text='RAISED', relief=tk.RAISED).pack()
tk.Button(root, text='RAISED', relief=tk.RIDGE).pack()
tk.Button(root, text='SOLID', relief=tk.SOLID).pack()
tk.Button(root, text='SUNKEN', relief=tk.SUNKEN).pack()

root.mainloop()
```

这段代码的运行效果如图 10.3 所示。

当按下某个按钮时,Label 中的文本便会改变,效果如图 10.4 所示。

3. RadioButton 单选按钮

RadioButton 是单选按钮,它具有组的概念。variable 属性是 RadioButton 所负责修改的变量,可以使用 Tkinter 模块的 IntVar 等类型,variable 变量相同的 RadioButton 属于同一组,不同组的 RadioButton 对象互相不影响。同时 RadioButton 也具有 command 属性,当 RadioButton 被选中时会被执行。

RadioButton 的实例请见如下代码:

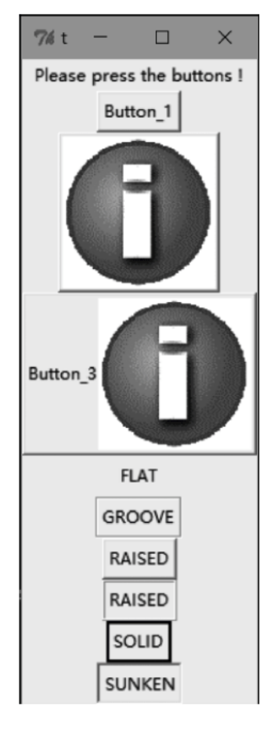


图 10.3　Button 组件

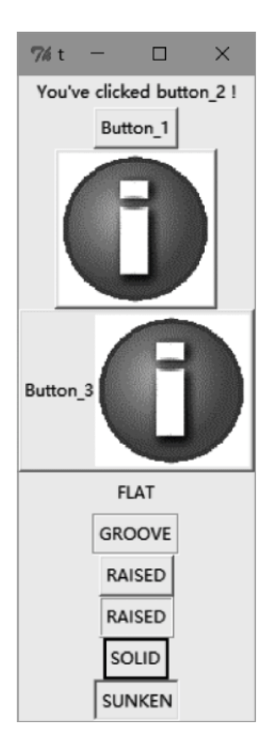


图 10.4　单击 Button_2 修改 Label 中的文本

```
# -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

# RadioButton 有组的概念,variable 为同一变量的 RadioButton 为一组,每组中最多只能有一个
#按钮被选中

tk.Label(root, text='Sex:').pack()

v_sex = tk.IntVar()
v_sex.set(0)

rb_s_b = tk.Radiobutton(root, text='BOY', variable=v_sex, value=0).pack()
rb_s_g = tk.Radiobutton(root, text='GIRL', variable=v_sex, value=1).pack()

# 不同组的 RadioButton 之前选择互相不影响

tk.Label(root, text='Class:').pack()

v_class = tk.IntVar()
```

```
v_class.set(1)

for rbnum in range(1, 6):
    tk.Radiobutton(root, text = str(rbnum), variable = v_class, value = rbnum).pack()

# RadioButton 也有 command 属性,当被选中时,command 的指令会被执行

label_favor = tk.Label(root, text = "What's your favorite fruit ?")
label_favor.pack()

def func_1():
    label_favor["text"] = 'Your favorite fruit is Apple !'
def func_2():
    label_favor["text"] = 'Your favorite fruit is Banana !'
def func_3():
    label_favor["text"] = 'Your favorite fruit is Pair !'

fruits = ["Apple", "Banana", "Pair"]
funcs = [func_1,func_2,func_3]

v_favor = tk.IntVar()
v_favor.set(-1)

for rbfnum in range(0,3):
    tk.Radiobutton(root,text = fruits[rbfnum],command = funcs[rbfnum],variable = v_favor,
value = rbfnum).pack()

root.mainloop()
```

上段代码的运行效果如图 10.5 所示。

当最后一组 RadioButton 被选中时,command 的函数会被执行,最后一个 Label 会被修改,效果如图 10.6 所示。

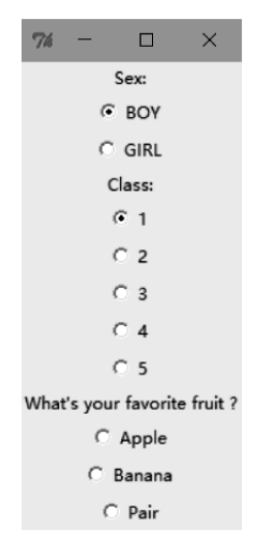


图 10.5　RadioButton 组件

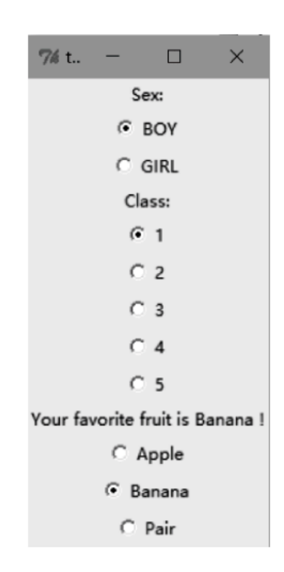


图 10.6　选择 RadioButton 改变 Label 文本

4. CheckButton 复选框

CheckButton 是复选框控件，与 RadioButton 类似，CheckButton 也具有 variable 属性，但 CheckButton 没有分组的概念，每个 CheckButton 对应一个变量。当变量类型为 IntVar 时，默认未选中时值为 0，选中时值为 1。同时 CheckButton 也可以自定义状态值，通过 onvalue、offvalue 变量可以设置状态值。CheckButton 的 command 属性在每次状态改变时都会执行。

我们通过下面这个例子学习 CheckButton 的使用方法。

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()

# CheckButton 不分组,每个 CheckButton 对应一个变量
# 当变量类型为 IntVar 时,默认未选中时为 0,选中时为 1
# CheckButton 也具有 command 属性,当其状态改变时调用

tk.Label(root,text = "Choose all the fruits you like !").pack()

label_1 = tk.Label(root)

fruits = ["Apple","Banana","Pair"]
vs_1 = [tk.IntVar(),tk.IntVar(),tk.IntVar()]

def updateState_1():
    label_1["text"] = ""
    for i in range(0,3):
        if vs_1[i].get() == 1 :
            label_1["text"] += "You've chosen " + fruits[i] + "\n"

label_1.pack()

for cbnum in range(0,3):
    tk.Checkbutton(root,variable = vs_1[cbnum],text = fruits[cbnum],command = updateState_
1).pack()

# CheckButton 状态值也可以通过 onvalue、offvalue 自定义

tk.Label(root,text = "Choose all the animals you like !").pack()

label_2 = tk.Label(root)

animals = ["Cat","Dog","Python"]
vs_2 = [tk.StringVar(),tk.StringVar(),tk.StringVar()]

for v in vs_2:
```

```
    v.set("")

def updateState_2():
    label_2["text"] = "You've chosen "
    for i in range(0,3):
        label_2["text"] += vs_2[i].get() + " "

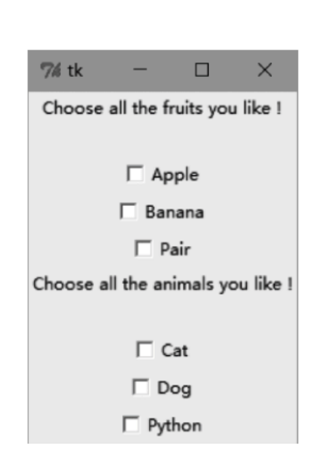
label_2.pack()

for cbnum in range(0,3):
    tk.Checkbutton(root, variable = vs_2[cbnum], text = animals[cbnum], onvalue = animals
[cbnum],offvalue = "",command = updateState_2).pack()


root.mainloop()
```

这段代码的运行效果如图 10.7 所示。

当 CheckButton 被选中时，Label 的文本会改变，效果如图 10.8 所示。



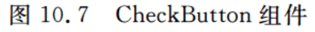


图 10.7　CheckButton 组件

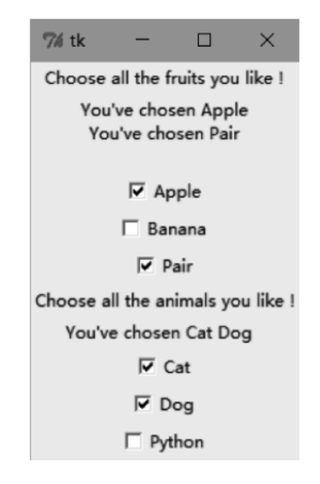


图 10.8　选中 CheckButton 改变 Label 中文本

5. OptionMenu 下拉选框

OptionMenu 是下拉选框，OptionMenu 的初始化函数接受多于两个参数，第一个为父对象，第二个为变量，其余的项为预选参数值。若想获取变量值，可以使用 get()方法。

OptionMenu 的使用见下例：

```
    # -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

# OptionMenu 的初始化函数接受多于两个参数，第一个为父对象，第二个为变量，其余的项为预选
#参数值
```

```
# 若想获取变量值,可以使用 get()方法

tk.Label(root,text = "请选择你的年级").pack()

v = tk.StringVar()
v.set("请选择")

tk.OptionMenu(root,v,"大一","大二","大三","大四").pack()

root.mainloop()
```

这段代码的运行效果如图 10.9 所示。

下拉菜单如图 10.10 所示。



图 10.9　收起的 OptionMenu



图 10.10　下拉后的 OptionMenu

6. Menu 菜单

Menu 是 Tkinter 提供的菜单组件。

Menu 与其他组件不同,Menu 不使用 pack()等方法显示,而是需要通过修改 root 窗体的 menu 属性为 Menu 对象来显示。

Menu 的菜单项可以使用 add_command 方法添加,该方法支持 command 属性。Menu 还支持子菜单,只需要使用 add_cascade 方法将子菜单绑定在父级菜单上即可。Menu 还可以通过 add_radiobutton、add_checkbutton 方法添加单选、多选按钮。add_separator 方法可以为菜单添加分隔符。

Menu 的 tearoff 属性用来控制菜单的裁剪功能的开与关。当 tearoff 为 1 时,可以单击菜单上的虚线将菜单从父级菜单中裁剪下来,单独成为一个菜单窗体。

我们通过下面的例子详细学习 Menu 的使用方法:

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()


def quit():
```

```
    root.quit()

menubar = tk.Menu(root)

# 使用 add_command 方法为 Menu 添加菜单项

menubar.add_command(label = "Quit", command = quit)

# 使用 add_cascade 方法为 Menu 添加子菜单

menu_fruits = tk.Menu(menubar, tearoff = 0)

for item in ["Apple", "Banana", "Pair"]:
    menu_fruits.add_command(label = item)

menubar.add_cascade(label = 'Fruits', menu = menu_fruits)

# 使用 add_radiobutton 为菜单添加单选框

menu_animals = tk.Menu(root, tearoff = 0)

v_animals = tk.IntVar()

for i, j in {'Cat': 0,
             'Dog': 1,
             'Monkey': 2}.items():
menu_animals.add_radiobutton(label = i, variable = v_animals, value = j)

menubar.add_cascade(label = 'Animals', menu = menu_animals)

# 使用 add_checkbutton 方法为 Menu 添加多选框

menu_foods = tk.Menu(menubar, tearoff = 0)

v_Bread = tk.IntVar()
v_Cake = tk.IntVar()
v_Rice = tk.IntVar()

for i, j in {'Bread': v_Bread,
             'Cake': v_Cake,
             'Rice': v_Rice}.items():
    menu_foods.add_checkbutton(label = i, variable = j)

menubar.add_cascade(label = 'Foods', menu = menu_foods)

# 使用 add_separator 为菜单添加分隔符
# 修改 tearoff 为 1,开启菜单裁剪功能
```

```
menu_languages = tk.Menu(menubar,tearoff = 1)

menu_languages.add_command(label = 'Python')
menu_languages.add_separator()
for i in ['C','Java','PHP']:
    menu_languages.add_command(label = i)

menubar.add_cascade(label = 'Languages',menu = menu_languages)

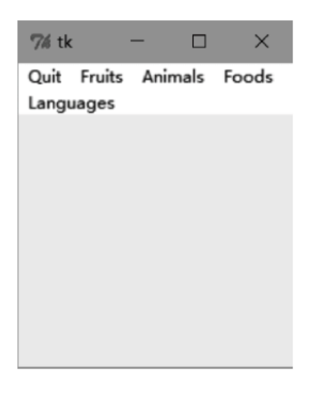
# 修改 root 的 menu 属性,显示 menubar

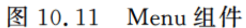
root['menu'] = menubar

root.mainloop()
```

这段代码的运行效果如图 10.11 所示,其中 Quit 为顶级菜单 menubar 的一个菜单项,其余均为 menubar 的子菜单。这段代码使用 root['menu'] = menubar 将 root 窗体的 menu 属性修改为 menubar,用来显示我们自定义的菜单。

Fruits 是 Menubar 的一个子菜单,效果如图 10.12 所示。




图 10.11　Menu 组件

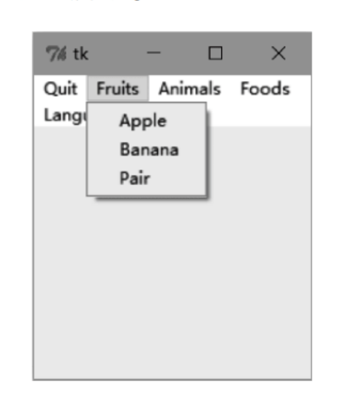


图 10.12　Menu 子菜单

Animals 菜单中按钮为单选按钮,效果如图 10.13 所示。

Foods 菜单中为多选按钮,效果如图 10.14 所示。

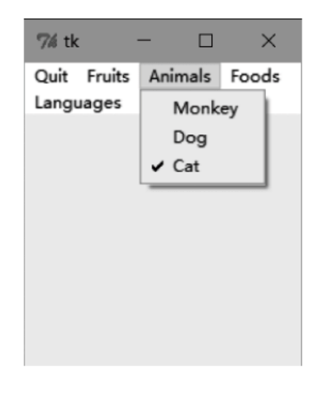


图 10.13　Menu 子单选组件

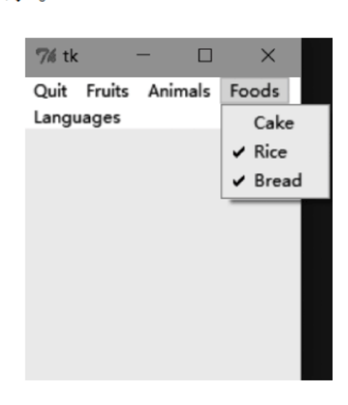
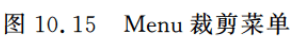


图 10.14　Menu 子多选组件

Languages 菜单中，Python 项与其余项之间添加了分隔符（如图实线）。Languages 菜单还开启了菜单裁剪功能，单击虚线即可裁剪菜单，如图 10.15 所示。

单击虚线裁剪菜单后，Languages 菜单单独成为一个窗体，效果如图 10.16 所示。




图 10.15　Menu 裁剪菜单

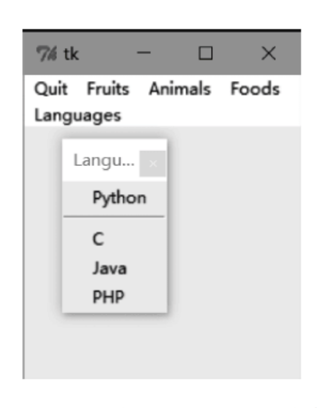


图 10.16　裁剪后的 Menu 窗体

7. Entry 输入框

```
# -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

# 使用 textvariable 指定 Entry 对象影响的文本变量

tk.Label(root,text = '请输入用户名:').pack()

v_user = tk.StringVar()
v_user.set("Python")

tk.Entry(root,textvariable = v_user).pack()

# 修改 Entry 的 show 属性,控制显示出来的内容,常用于密码输入

tk.Label(root,text = '请输入密码:').pack()

v_passwd = tk.StringVar()
v_passwd.set("123456")

tk.Entry(root,textvariable = v_passwd,show = '*').pack()

# 修改 Entry 的 state 为 readonly 使 Entry 中的内容只读

tk.Label(root,text = '下面这段文字是只读的:').pack()

v_readonly = tk.StringVar()
```

```
v_readonly.set("这段文字是只读的!")

tk.Entry(root,textvariable = v_readonly,state = 'readonly').pack()

root.mainloop()
```

这段代码的运行效果如图 10.17 所示。

8. SpinBox 选值框

SpinBox 是用来简单调整值的组件。

SpinBox 的 textvariable 属性指向影响的变量,from_与 to 分别是变量的最小、最大值,increment 为步距,即每次单击上下箭头的增量。

此外,SpinBox 还可以通过定制 values 自定义取值。我们可以通过修改 values 实现无规则地跳跃取值。

SpinBox 也具有 command 属性,每次修改值时会被调用。



图 10.17　Entry 组件

SpinBox 的实例代码如下:

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()

# SpinBox 的 textvariable 属性指向影响的变量
# from_与 to 分别是变量的最小、最大值
# increment 为步距,即每次单击上下箭头的增量

v_1 = tk.IntVar()

tk.Spinbox(root, textvariable = v_1, from_ = 0, to = 100, increment = 5).pack()

# SpinBox 还可以通过修改 values 来自定义可选值

v_2 = tk.IntVar()

tk.Spinbox(root, textvariable = v_2, values = (0, 10, 50, 100, 200, 500)).pack()

root.mainloop()
```

这段代码的运行效果如图 10.18 所示。

当两个选框分别单击一次上箭头后,效果如图 10.19 所示。



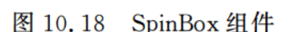


图 10.18　SpinBox 组件



图 10.19　单击一次上箭头后的值

9. Scale 选值条

Scale 是通过拖曳选值的条形组件。

Scale 的 variable 属性指向影响的变量,from_与 to 分别为最小、最大值。resolution 属性控制 Scale 的最小步长,该值默认为 1。

Scale 还有一个特有的属性 orient,该属性用于控制 Scale 控件的横纵方向。

我们通过下面的例子学习 Scale 组件的使用方法:

```
# -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

# Scale 的 variable 属性指向影响的变量
# from_与 to 分别为最小、最大值
# orient 属性用来控制 Scale 组件的横纵方向

v_1 = tk.IntVar()

tk.Scale(root, from_=0, to=100, variable=v_1).pack()
tk.Scale(root, from_=0, to=100, variable=v_1, orient=tk.HORIZONTAL).pack()

# resolution 属性控制 Scale 的最小步长,该值默认为 1

v_2 = tk.DoubleVar()

tk.Scale(root, from_=0, to=100, variable=v_2,orient=tk.HORIZONTAL, resolution=0.1).
pack()


root.mainloop()
```

这段代码的运行效果如图 10.20 所示。

10.2.3 Tkinter 布局

10.2.2 节中我们学习了 Tkinter 各种常用组件的用法,然而,之前我们只是将各种组件堆叠在一起。一个优秀的 GUI 程序除了使用组件实现功能外,还要有一个简洁易用的布局来排列这些组件,本节中,我们将讲解 Tkinter 常用的布局管理模式。

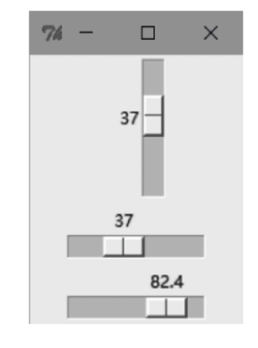


图 10.20 Scale 组件

1. pack + Frame

在之前的例子中,我们一直在使用 pack 默认参数进行布局,其实 pack 还支持很多布局方式。pack 方法在布局中重要的参数有 side、expand 和 fill。

side 参数可以取 Tkinter.LEFT、Tkinter.RIGHT、Tkinter.TOP、Tkinter.BOTTOM,分别为从左、右、上、下加入显示。

expand 参数可以取 Tkinter. YES 与 Tkinter. NO，当 expand 开启时，允许该组件占用尽可能多的空白(这一概念可能不太好理解，我们将通过一个例子进行讲解)。

fill 参数决定组件的填充方向，默认为 Tkinter. NONE，即不填充，只占用足够容纳自身内容的大小。fill 参数还可以取 Tkinter. X、Tkinter. Y、Tkinter. BOTH，分别表示允许沿 X 或 Y 或同时沿 XY 方向填充。

我们通过下面的例子学习这三个参数的使用与区别：

```
# -*- coding: utf-8 -*-

import Tkinter as tk

# r0 窗体中，红色与蓝色从左侧加入显示，绿色从右侧加入显示

root_0 = tk.Tk()
root_0.title('r0')
# TK 对象的 geometry 用来设置窗口大小
root_0.geometry('200x200')

tk.Label(root_0, text = 'Label', bg = 'red').pack(side = tk.LEFT)
tk.Label(root_0, text = 'Label', bg = 'green').pack(side = tk.RIGHT)
tk.Label(root_0, text = 'Label', bg = 'blue').pack(side = tk.LEFT)

# r1 中，Label_1 没有开启 expand

root_1 = tk.Tk()
root_1.title('r1')
root_1.geometry('200x200')

tk.Label(root_1, text = 'Label_1', bg = 'green').pack(expand = tk.NO, fill = tk.Y)
tk.Label(root_1, text = 'Label_2', bg = 'red').pack(fill = tk.BOTH)

# r1 中，Label_1 开启了 expand

root_2 = tk.Tk()
root_2.title('r2')
root_2.geometry('200x200')

tk.Label(root_2, text = 'Label_1', bg = 'green').pack(expand = tk.YES, fill = tk.Y)
tk.Label(root_2, text = 'Label_2', bg = 'red').pack(fill = tk.BOTH)


tk.mainloop()
```

这段代码运行会得到三个窗体，如图 10.21 和图 10.22 所示。



图 10.21　pack 布局



图 10.22　使用 fill 或 expand 属性

窗体 r0 中，我们研究 side 属性，将红色与蓝色从左侧加入显示，绿色从右侧加入显示，得到图 10.21 所示的效果。

窗体 r1、r2 用来观察 fill 属性与 expand 属性的效果。r1 中，Label_1 没有开启 expand 属性，虽然 fill 属性使其向 Y 轴方向填充，也不会占用比自己更多的空间；而 r2 中的 Label_1 开启了 expand 属性，因此 Label_1 占用了尽可能多的空白空间。两例中 Label_2 均同时向 XY 两轴填充。

单独使用 pack 进行布局时，难以应付复杂的布局方式。因此，我们可以使用 Tkinter 中提供的 Frame 组件。Frame 组件是一个容器类型的组件，它用来容纳子级组件，我们可以将 Frame 嵌套实现更复杂的效果，便于布局与 Frame 的重用。

下面通过下例来使用 Frame+pack 的方式实现稍复杂的布局：

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()

root.geometry('200x200')

tk.Label(root,text = 'A',bg = 'green').pack(side = tk.TOP,fill = tk.X)

frame_bottom = tk.Frame(root)
frame_bottom.pack(side = tk.TOP,expand = tk.YES,fill = tk.BOTH)

frame_b_left = tk.Frame(frame_bottom)
frame_b_left.pack(side = tk.LEFT,expand = tk.YES,fill = tk.BOTH)

tk.Label(frame_bottom,text = 'D',bg = 'red').pack(side = tk.LEFT,fill = tk.Y)

tk.Label(frame_b_left,text = 'B',bg = 'blue').pack(side = tk.TOP,expand = tk.YES,fill = tk.
BOTH)
tk.Label(frame_b_left,text = 'C',bg = 'yellow').pack(side = tk.TOP,fill = tk.X)

root.mainloop()
```

这段代码的运行效果如图 10.23 所示。

2. PanedWindow

为了方便地对组件进行布局，Tkinter 引入了 PanedWindow 组件。PanedWindow 是一个容器组件，只用来容纳其他组件。PanedWindow 可以创建格子式的布局。PanedWindow 格子方向默认为横向，通过修改 orient 属性，还可以使用纵向布局。

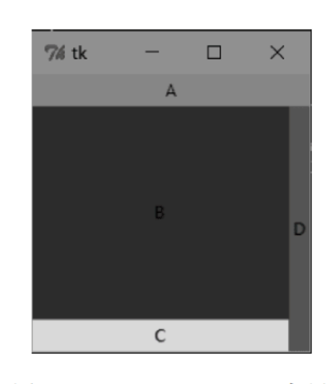


图 10.23 pack＋Frame 布局

PanedWindow 组件支持嵌套，可以使用 add 方法将其他组件(当然也包括 PanedWindow)加入 PanedWindow 的格子。通过不断的嵌套，可以实现各式各样的布局。

PanedWindow 的另一个好处是用户可以通过拖曳格子之间的间隙调整每块格子的大小，适应各种习惯的用户。

PanedWindow 既可以像之前的例子一样作为组件加入顶级窗体中，也可以直接用 PanedWindow 代替顶级窗体，本节的例子中我们便会这样做。

```
# -*- coding: utf-8 -*-

import Tkinter as tk

'''

#依旧可以这种方式使用 PanedWindow,不过本例我们使用一种更加方便的方式

root = tk.Tk()

pw_root = tk.PanedWindow(root)
pw_root.pack()

root.mainloop()

'''

# 直接将 PanedWindow 作为一个顶级窗体

pw_root = tk.PanedWindow()

# 将 PanedWindow 使用 pack 的方式显示

pw_root.pack()

# PanedWindow 的 add 方法可为其新增一格,格子中可以容纳其他组件

label_1 = tk.Label(pw_root, text = 'Left',bg = 'green')
pw_root.add(label_1)
```

```
# PanedWindow 可以嵌套另一个 PanedWindow,实现更丰富的效果
# PanedWindow 默认方向为横向,可以通过 orient 属性修改

pw_1 = tk.PanedWindow(pw_root, orient = tk.VERTICAL)
pw_1.add(tk.Label(pw_1, text = 'TOP',bg = 'blue'))
pw_1.add(tk.Label(pw_1, text = 'BOTTOM',bg = 'yellow'))

pw_root.add(pw_1)

# 直接调用顶级 PanedWindow 的 mainloop 方法开始事件循环

pw_root.mainloop()
```

这段代码的运行效果如图 10.24 所示。

如图 10.24 所示,顶级 PanedWindow 为默认的横向,我们分别向其中加入了一个 Label 和一个子 PanedWindow;子 PanedWindow 被设置为竖向,分别添加了两个 Label 组件。

当鼠标指针放在 PanedWindow 格子之间时,指针会变为箭头,此时我们可以拖曳调整格子大小,如图 10.25 所示。

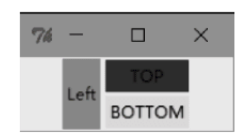


图 10.24　PanedWindow 嵌套



图 10.25　拖曳调整大小

3. place

place 方法是 Tkinter 提供的另一种组件布局方式,place 方法可以通过坐标、比例或者二者结合的方式来显示组件。

使用坐标时,通过修改 x、y(原点为父级容器的位置)参数定位;使用比例时,通过修改 relx、rely(均相对于父级容器)参数定位;混合使用坐标与比例时,会先根据坐标计算出初始位置,再偏移坐标位置来得到最终的位置。

place 的 anchor 参数用来改变组件显示的对齐方式。anchor 默认采用 CENTER,还可以使用 N、S、W、E、NW、SW、NE、SE、NS、EW、NSEW(NSWE 即北南西东)的方式对齐,读者可以自己实践体会这些对齐方式的区别,这里不再赘述。

place 的主要用法详见本例:

```
# - * - coding: utf - 8 - * -

import Tkinter as tk

root = tk.Tk()
root.geometry('300x200')
```

```
# 使用坐标(原点相对于父级容器)
tk.Label(root, text = "This is a Label !").place(x = 150, y = 50, anchor = tk.CENTER)
# 使用比例(相对于父级容器)
tk.Label(root,text = 'This is a Label, too !').place(relx = 0.5,rely = 0.5,anchor = tk.CENTER)
# 结合使用坐标与比例时,先按照比例计算位置,再偏移坐标的位置
tk.Label(root,text = 'This is also a Label !').place(relx = 0.5,rely = 0.5,x = 50,y = 50,anchor
 = tk.CENTER)
tk.mainloop()
```

这段代码的运行效果如图 10.26 所示。



图 10.26　place 布局

4. grid

以上三种方式虽然已经可以实现各种 GUI 样式,然而均有复杂的缺点。本节我们将讲解另一种灵活、便捷的 GUI 布局方式——grid 网格布局。

顾名思义,grid 布局管理器使用网格的方式,具有行、列的概念,可以使用 grid 像填表一样完成对组件的快速布局。

使用 grid 时,需要修改 row 与 column 属性,它们分别表示组件要被放置的行、列。grid 的行、列均以 0 起始,空行、列在渲染时会被忽略(例如 0、2 行有组件而 1 行为空,则 0、2 行之间不会有一行空隙)。

如果一个组件需要跨行、列放置,只需修改 rowspan、columnspan 属性为需要跨的行、列数即可。

grid 的 sticky 属性控制组件的对齐方式,我们可以使用 N、S、W、E 及其一些组合来修改,默认为 NSEW(居中)。

注意,在同一父容器下,不要同时使用 pack 和 grid。

本例中我们将使用 grid 完成一个简易的登录 GUI 的程序:

```
# -*- coding: utf-8 -*-

import Tkinter as tk

root = tk.Tk()

# grid 的 row、column 属性分别控制放置的行、列
# grid 的行列均从 0 开始

tk.Label(root, text='username:').grid(row=1, column=0)
tk.Label(root, text='password:').grid(row=2, column=0)

v_user = tk.StringVar()
v_passwd = tk.StringVar()

tk.Entry(root,textvariable=v_user).grid(row=1,column=1)
tk.Entry(root,textvariable=v_passwd,show='*').grid(row=2,column=1)

# rowspan、columnspan 属性控制组件跨行、跨列显示

tk.Label(root,text='Welcome !').grid(row=0,column=0,columnspan=2)

img = tk.PhotoImage(file='res/img_login.gif')
tk.Label(root,image=img).grid(row=0,column=2,rowspan=3)

# sticky 属性控制内容的对齐方向,允许使用 N、S、W、E 及其一些组合,默认为 NSEW 居中

tk.Label(root, text="How convenient 'grid' is !", bg='red').grid(row=3, column=0,
columnspan=3,sticky=tk.E)

tk.mainloop()
```

这段代码的运行效果如图 10.27 所示。

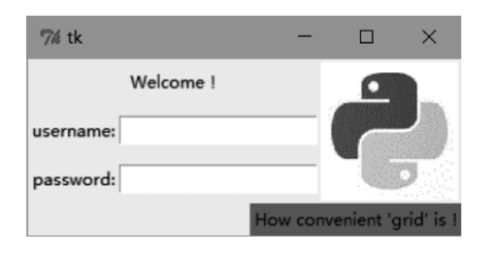


图 10.27　grid 布局

10.3　使用 Tkinter 模块编写 GUI 程序

学习了前面的章节,我们已经可以编写 GUI 程序了。但是当程序的逻辑功能、GUI 都比较复杂时,松散的代码往往不利于维护。本节我们将讲解如何安排 Tkinter GUI 的代码

风格，写出清晰易懂的 Tkinter GUI 程序。

10.3.1 Tkinter GUI 封装

模块化的代码利于拓展、维护，本节中，我们将把 Tkinter GUI 的具体实现封装在 App 类中。之前的章节为了让读者区分 Tkinter 的组件，将 Tkinter 引入时重命名为 tk 便于区分，本节开始，为了简洁起见，我们将直接引入 Tkinter 下的所有内容。

先观察如下代码：

```
# -*- coding: utf-8 -*-

from Tkinter import *


class App:

    def __init__(self, master):

        self.master = master

        frame = Frame(master)
        frame.pack()

        Label(frame, text="Hellow World !").grid(columnspan=2)

        Label(frame, text="Click to Quit !").grid(row=1, column=0)

        self.button_quit = Button(frame, text="Quit", command=self.quit)
        self.button_quit.grid(row=1, column=1)

    def quit(self):
        self.master.quit()
        self.master.destroy()

if __name__ == '__main__':

    root = Tk()

    app = App(root)

    root.mainloop()
```

在这段代码中，我们将 GUI 的具体实现封装在了 App 类中，在主程序中仅新建了 root 窗体，创建 App 类的实例 app 并与 root 窗体绑定，最后开始 root 窗体的事件循环。同时，我们在程序入口加入了对程序模块的检测，只有当本程序作为入口时才能执行，避免程序被错误引用。这种封装方式十分简洁，也方便以后的修改，在真正开发 GUI 应用时，我们推荐采取这种方式封装我们的 App。

App 的初始化函数需要传递父级容器(本例中即为顶级窗体 root),并且在初始化函数中实现了具体的 GUI。当 GUI 组件需要调用程序的逻辑代码时,我们只需将逻辑代码封装为 App 类的一个函数,再将 command 属性赋为这个函数即可。除此之外,我们在 quit 窗体时还加入了 destory 释放其占用的内存空间。

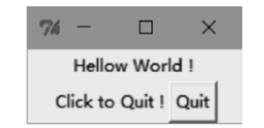
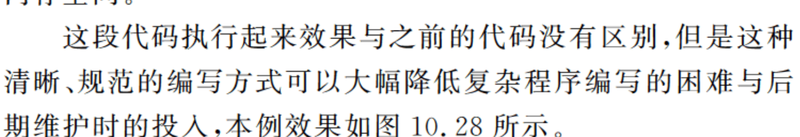


图 10.28　Tkinter 封装测试



这段代码执行起来效果与之前的代码没有区别,但是这种清晰、规范的编写方式可以大幅降低复杂程序编写的困难与后期维护时的投入,本例效果如图 10.28 所示。

10.3.2　Tkinter 事件

GUI 程序不仅可以通过操作组件进行控制,有时还需要使用鼠标、键盘等帮助控制(例如定制热键等),这时,我们便需要使用事件绑定等方式实现功能。

1. focus_set 设置焦点

focus_set 方法用于设置程序焦点,例如一个 GUI 窗体有三个输入框,我们将焦点设置在第二个输入框上,当 GUI 窗体渲染完成时,光标便会在第二个输入框中闪烁,此时便可直接在其中输入文本。

如本例中,我们将焦点设置在了 entry_2 上:

```
# -*- coding: utf-8 -*-

from Tkinter import *


class App:

    def __init__(self, master):

        self.master = master

        frame = Frame(master)
        frame.pack()

        label_1 = Label(frame,text = "Entry_1:")
        entry_1 = Entry(frame)

        label_2 = Label(frame,text = "Entry_2:")
        entry_2 = Entry(frame)

        label_3 = Label(frame,text = "Entry_3:")
        entry_3 = Entry(frame)

        label_1.grid(row = 0,column = 0)
        entry_1.grid(row = 0,column = 1)

        label_2.grid(row = 1,column = 0)
```

```
        entry_2.grid(row = 1,column = 1)
        entry_2.focus_set()

        label_3.grid(row = 2,column = 0)
        entry_3.grid(row = 2,column = 1)

if __name__ == '__main__':

    root = Tk()

    app = App(root)

    root.mainloop()
```

本例执行效果如图 10.29 所示。

图 10.29 中可见光标在 Entry_2 对应的输入框内闪烁。

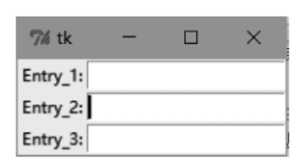


图 10.29　修改焦点

2. bind 事件绑定

当我们需要捕捉其他事件时，就要使用 bind，bind 方法用于为组件绑定事件，我们可以使用如下方式进行绑定：

```
组件.bind(事件,方法)
组件.bind_all(事件,方法)
```

bind 与 bind_all 的区别在于：使用 bind 绑定，只对当前组件绑定，在键盘事件中，只有焦点在该组件上，对应的方法才会响应；bind_all 相当于对所有组件绑定。

bind 方法可以绑定各种 Tkinter 事件。虽然事件种类繁多，但拥有统一的模板：

```
<[modifier - ]...type[ - detail]>
```

Tkinter 的事件被"<>"包围，其中最主要的属性是 type，用来表示事件类型，包括键盘事件、鼠标事件等；modifier 用于增加组合键如 Control、Shift 等；detail 用于具体的事件描述。

例如：

(1) <Button-1>表示鼠标左键；

(2) <KeyPress-A>表示键盘 A 键；

(3) <Control-Shift-A>表示同时按下 Ctrl、Shift、A 键。

Tkinter 支持的事件有很多类型(type)，详见表 10.5。

表 10.5　Tkinter 事件类型

事　　件	描　　述
Activate	组件从非激活状态到激活状态
Button	按下鼠标按键
ButtonRelease	松开鼠标按键

续表

事　件	描　述
Configure	组件尺寸改变
Deactivate	组件从激活状态到非激活状态
Destroy	组件被销毁时
Enter	鼠标移动到组件上
Expose	窗口被遮蔽后重现时
FocusIn	组件获得焦点时
FocusOut	组件失去焦点时
KeyPress	键盘按键按下时
KeyRelease	键盘按键松开时
Leave	鼠标从组件上移出时
Map	组件变为可见时
Motion	鼠标在组件上移动时
MouseWheel	滑动滚轮时
Unmap	组件变为不可见时
Visibility	窗口从完全遮蔽到部分可见时

Tkinter 的事件修饰语(modifier)详见表 10.6。

表 10.6　Tkinter 事件修饰语

修　饰　语	描　述
Alt	按下 Alt 键时
Any	按下任意按键时
Control	按下 Control 时
Double	双击时
Shift	按下 Shift 时
Triple	三击时

注：鼠标相关事件中,1 代表左键,2 代表中键,3 代表右键。

使用 bind 绑定的方法,需要接受一个 event 对象作为参数。event 对象中包括对事件响应的各种参数,详见表 10.7。

表 10.7　event 对象参数

event 参数	描　述
char	键盘事件中按下的按键
delta	鼠标滚轮滚动大小
width、height	窗体尺寸改变后的新宽、新高
keycode	数字按键值
keysym	特殊按键值
num	鼠标按键
widget	触发事件的组件
x、y	鼠标相对于触发事件的组件的坐标
x_root、y_root	鼠标相对于屏幕左上角的坐标

下面我们通过一个例子学习 bind 的使用方法：

```
# -*- coding: utf-8 -*-

from Tkinter import *


class App:

    def __init__(self, master):

        self.master = master

        frame = Frame(master)
        frame.pack()

        label_green = Label(frame, text='Green', bg='green')
        label_blue = Label(frame, text='Blue', bg='blue')
        label_red = Label(frame, text='Red', bg='red')
        label_yellow = Label(frame, text='Yellow', bg='yellow')

        label_green.pack(side=TOP, fill=BOTH)
        label_blue.pack(side=LEFT, fill=BOTH)
        label_red.pack(side=BOTTOM, fill=BOTH)
        label_yellow.pack(side=RIGHT, fill=BOTH)

        label_yellow.bind("<Enter>", self.event_1)
        # label_yellow.bind_all("<Enter>",self.event_2)

    def event_1(self, event):
        print event.widget['text'],event.x,event.y

    def event_2(self, event):
        print "event2:", event.x, event.y

if __name__ == '__main__':

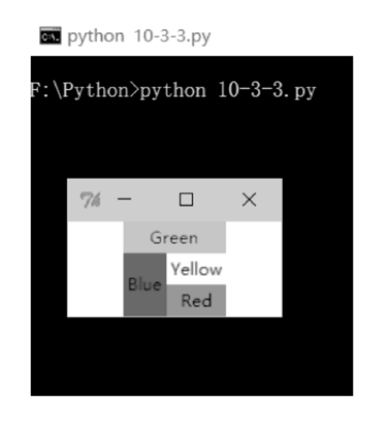
    root = Tk()

    app = App(root)

    root.mainloop()
```

在本例中，我们放置了 4 个 Label，并且为 label_yellow 绑定了鼠标移入事件，程序执行效果如图 10.30 所示。

当鼠标指针移动到 label_yellow 上时，控制台会输出 text、x、y 的值，如图 10.31 所示。



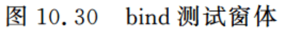


图 10.30　bind 测试窗体

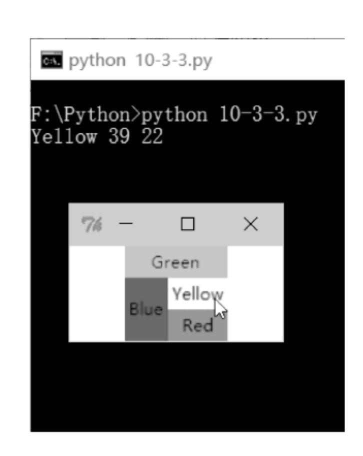


图 10.31　特定组件执行 bind 事件

而当使用 bind_all 方法时，无论指针进入哪个组件，绑定的方法都会执行，如图 10.32 所示。



图 10.32　任意组件执行 bind 事件

习　题　10

一、选择题

1. Tkinter 组件的(　　)属性用于调节背景颜色。

   A. background　　B. foreground　　C. scale　　D. color

2. Tkinter 中可拉动的范围组件是(　　)。

   A. Label　　B. Table　　C. Scale　　D. Tool

3. Tkinter(　　)布局采用网格的模式布局。

   A. place　　B. pack+Frame　　C. PanedWindow　　D. grid

二、填空题

1. Tkinter 中，________组件用来添加标签，________组件用来添加按钮。
2. Tkinter 中，单选按钮与多选按钮的组件分别是________与________。
3. 在编写 Tkinter GUI 脚本最后，我们需要使用________函数启动 GUI 事件循环。

4. Tkinter 中的通用属性是指________。

5. Tkinter 使用________函数绑定事件。

三、论述题

简述 Tkinter 的布局方式,并解释每种布局的适用场景。

四、编程题

QQ 是中国最常用的即时通信软件之一,请使用 Tkinter 模拟 QQ 登录布局。

第11章 Python 多线程与多进程编程

多线程与多进程是编写实用程序的必要方式。无论是 GUI、并发网络通信、并发缓存读写，都需要用到多线程与多进程技术。

本章我们将介绍线程与进程的概念、Python 多线程与多进程的实现以及 Python 的多线程与多进程的特性，这些都是未来使用 Python 编写复杂程序必不可少的知识。

11.1 线程与进程

在介绍多线程与多进程之前，我们先了解什么是线程，什么又是进程。

11.1.1 进程

虽然本章将先介绍多线程编程再介绍多进程编程，但是为了理解二者的概念，我们还是要从进程开始讲起。

进程(Process)是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体。

进程的特性可以大致概括为以下几个方面：

- 动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。
- 并发性：任何进程都可以同其他进程一起并发执行。
- 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。
- 结构特征：进程由程序、数据和进程控制块三部分组成。

通俗来说，一个正在运行的程序即是一个进程。虽然多进程已经可以实现并发程序，然而，进程的开销是相对较大的，而且不同进程之间可共享的内容也是很局限的。随着计算机技术的发展，一种开销更小、相互共享内容更多的技术应运而生，那就是线程。

11.1.2 线程

线程是操作系统能够进行运算调度的最小单位(程序执行流的最小单元)。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个

进程中可以并发多个线程，每条线程并行执行不同的任务。

线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。

11.1.3 多线程与多进程

多线程与多进程在现代计算机中已经是不可或缺的技术。

例如我们的带有 GUI 的视频播放器程序，我们可以通过鼠标的输入来控制播放或者停止，但是在播放状态下，即使我们不使用鼠标，视频仍会播放下去，而不需要等待我们的鼠标输入。这样的例子在绝大多数 GUI 程序中都成立，虽然我们对此已经习以为常，但是这仍是通过简单的循环实现不了的。网络编程更是如此，一个服务端程序可能同时与数十个客户端程序通过网络通信，如果一个客户端程序的网络环境不佳，我们不希望因为它网络速度阻塞其他客户端的网络通信，这也需要通过多线程来实现。

多进程同样十分常见。例如，我们可以一边下载软件、一边看电影，此时，下载器和电影播放软件即为两个进程。因为进程具有并发性，因此我们可以在看电影的同时下载软件。

11.2 Python 多线程编程

既然多线程在 GUI、网络通信、耗时运算等方面如此常用，我们就来学习 Python 的多进程编程。

11.2.1 Python 多线程的特殊性

在真正介绍 Python 多线程编程之前，我们必须了解 Python 多线程的特殊性。

细心的读者可能已经发现，在这里我们没有使用“特性”这个字眼，而是使用“特殊性”，这是因为 Python 多线程的特殊性并不是由 Python 语言本身产生的，而是 Python 的一些解释器产生的。有些解释器为了保证线程安全(线程安全问题我们将在下面详细讲解，在此读者可以简单理解为多个线程同时访问并修改同一资源会引起的问题)而引入了 GIL (Global Interpreter Lock，全局解释器锁)。而由于 GIL 的存在，同一时间解释器只能解释一条字节码(bytecode)。

Python 的解释器 CPython 就引入了 GIL，而我们大多数运行 Python 的环境即为 CPython，因此有时我们会说“Python 的多线程并不是真正的多线程”，其实在其他一些 Python 解释器下，如 JPython，其因为没有 GIL 限制，其多线程即为真正的多线程。因此，我们不能将这一缺陷总结为 Python 的语言特性，而是我们大多数运行 Python 的环境的特性。

既然在我们的运行环境中 Python 的多线程不会同时执行多条指令，即其并非真正并发执行，那么我们在这种情况下使用多线程还有意义吗？显然，这个问题的答案是肯定的。即使我们无法在此环境下通过同时多线程提高程序的运算速度(其实由于线程之间的切换，GIL 下的多线程运算速度反而比单线程要慢)，但是对于 IO 密集型程序，这种多线程仍然可以减少单线程的阻塞时间。例如，我们需要从网络中保存 10 个来自不同网页上的数据，

如果使用单线程，我们只有等待一个网页数据全部加载完后才能加载下一个网页中的内容，而网络的传输速度往往相对其他操作速度慢很多，因此单线程此时便会因为网络环境而阻塞变慢；而如果使用多线程，我们可以同时加载10个不同页面中的内容，因为每个页面可能来自不同网络，彼此间速度的竞争可能没有那么激烈，因此我们可以在其他页面加载时保存已经加载完毕的页面中的数据，这样便提高了IO密集型程序的效率。

11.2.2 使用 threading 模块进行多线程编程

Python 自带模块 threading 用来实现多线程编程。在使用前，先用 import threading 来导入 threading 模块。

1. 创建与运行线程

使用 threading 模块创建线程常见的，一种方式是直接使用 threading 模块的 Thread()方法创建并初始化一个线程对象，Thread 常用的参数有 target 与 args，target 为线程调用的函数，args 为该函数需要的参数，以元组的方式传递。实例化线程对象后，我们可以使用成员函数 start()运行该线程：

```
# -*- coding: utf-8 -*-

import threading
import time

def func(id):
    print "Thread %d started !\n" % id
    if(id == 1):
        time.sleep(2)
    print "Thread %d finished !\n" % id

t1 = threading.Thread(target = func,args = (1,))
t2 = threading.Thread(target = func,args = (2,))

t1.start()
t2.start()
```

这段代码的运行效果如图11.1所示。

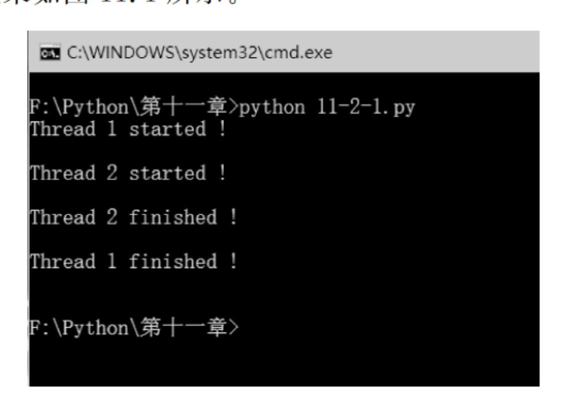


图11.1 创建与运行线程

我们可以看到，当调用 start()方法时，线程对象绑定的 target 函数便开始执行，而且两个线程对应的函数并发执行，互不干扰。因为在 func 函数中，我们设置线程 1 在开始后延迟 2 秒结束，而线程 2 开始直接结束，因此会得到图 11.1 中的输出顺序。如果不使用多线程而直接调用 func(1)、func(2)，我们将得到如图 11.2 所示的输出。

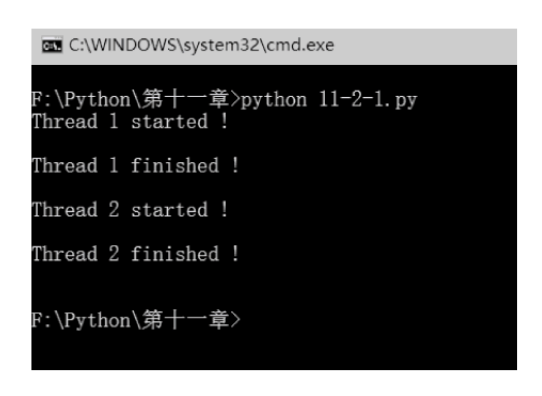


图 11.2　单线程测试

除了使用 threading 的 Thread()方法返回线程实例外，我们还可以自定义线程类型并使其继承 threading.Thread 类，此时，我们需要在自定义类的__init__函数中运行 Thread 类的__init__函数并重写 run 方法作为线程执行的函数。再实例化自定义线程类型并使用 start()方法运行线程：

```
import threading
import time


class MyThread(threading.Thread):

    def __init__(self, id):
        threading.Thread.__init__(self)
        self.id = id

    def run(self):
        print "Thread %d started !\n" % self.id
        if(self.id == 1):
            time.sleep(2)
        print "Thread %d finished !\n" % self.id

t1 = MyThread(1)
t2 = MyThread(2)

t1.start()
t2.start()
```

这段代码的运行效果与之前的多线程代码相同，如图 11.3 所示。

2. 使用 join()函数阻塞线程

在某一线程中对已经开始的线程使用 join()方法，可以阻塞当前线程，直到被 join()的

线程执行完后，被阻塞的线程才能继续执行。如下这段代码便演示了 join()方法：

```
C:\WINDOWS\system32\cmd.exe

F:\Python\第十一章>python 11-2-2.py
Thread 1 started !

Thread 2 started !

Thread 2 finished !

Thread 1 finished !


F:\Python\第十一章>
```

图 11.3　另一种创建线程的方式

```
import threading
import time


def func(sleeptime):
    time.sleep(sleeptime)
    print "Thread which slept %d second finished !\n" % sleeptime

t1 = threading.Thread(target=func, args=(1,))
t2 = threading.Thread(target=func, args=(2,))

print "All Threads start at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime
(time.time())) + "\n"

t1.start()
t2.start()

t1.join()

print "Now is : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time
())) + "\n"
```

运行这段代码，我们会得到如图 11.4 所示的输出。

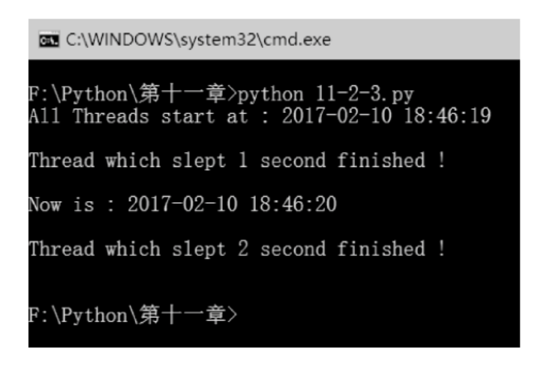


图 11.4　使用 join

从图 11.4 中可见，当线程 t1、t2 开始执行后，我们在主线程中对 t1 使用 join()方法，主线程中第二句输出在 t1 线程执行完后才能执行；t2 线程则正常等待 2 秒后退出。

join()方法还可以传递参数设置超时时间，即当被 join 的线程如果过了这个时间还没有执行完，则当前线程不会再等待被 join 的线程而直接开始执行：

```
import threading
import time

def func(sleeptime):
    time.sleep(sleeptime)
    print "Thread which slept %d second finished !\n" % sleeptime

t3 = threading.Thread(target = func, args = (3,))


print "All Threads start at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime
(time.time())) + "\n"

t3.start()

t3.join(2)

print "Now is : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time
())) + "\n"
```

这段代码的运行效果如图 11.5 所示。

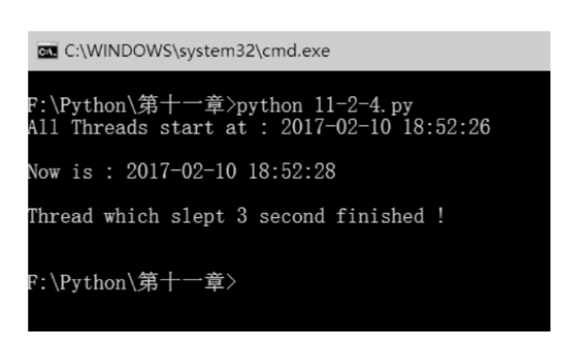


图 11.5　设置 join 延时

在这段代码中，t3 将在 3 秒后退出，而我们在主线程中对 t3 使用 join()方法阻塞主线程并设置 t3 的超时时间为 2 秒，因此在 2 秒后，主线程先继续执行，再过 1 秒后 t3 线程再输出并退出。

3. 守护线程

在 Python 中，主线程结束后，非守护线程仍会执行直到其结束；而守护线程则会在主线程结束后被终止。

Python 中守护线程的创建非常简单，只需要在线程开始前调用相应对象的 setDaemon(True)方法即可：

```
import threading
import time

def func():
    print "Thread started !"
    time.sleep(5)
    print "Thread finished !"

print "Main Thread started at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time())) + "\n"

t = threading.Thread(target = func)
t.setDaemon(True)
t.start()

time.sleep(2)
print "Main Thread finished at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time())) + "\n"
```

这段代码的运行效果如图 11.6 所示。



图 11.6 守护线程

在上面这段代码中，子线程被设置为守护线程，在子线程开始时的输出正常被打印，随后子线程会挂起 5 秒钟，而主线程在子线程开始 2 秒后结束。因为子线程为守护线程，在主线程结束后立刻被终止，因此子线程第二段输出因为已经被提前终止而没有被打印。

4. 线程安全与线程锁

在 Python 中，在子线程中，可以使用 global 方法访问全局变量。然而，不同线程在同时操作同一个对象时，可能会产生线程安全问题。我们将通过下面这个例子演示线程不安全的代码：

```
import threading
import time

def decNum():
    global num
    time.sleep(1)
```

```
        num -= 1

num = 100
thread_list = []

for i in range(100):
    t = threading.Thread(target = decNum)
    t.start()
    thread_list.append(t)

for t in thread_list:
    t.join()


print('final num:', num)
```

在上面这段代码中，我们初始化全局变量 num 为 100，然后创建了 100 个子线程，每个线程执行的代码都是挂起 1 秒再使 num 自减 1。我们阻塞主线程，使其在 100 个子线程结束后再输出 num 的值。在 100 个线程执行完毕后，num 被自减了 100 次，最终的输出应该是 0，那么，事实是这样的吗？我们来看多次执行这段代码的结果，如图 11.7 所示。

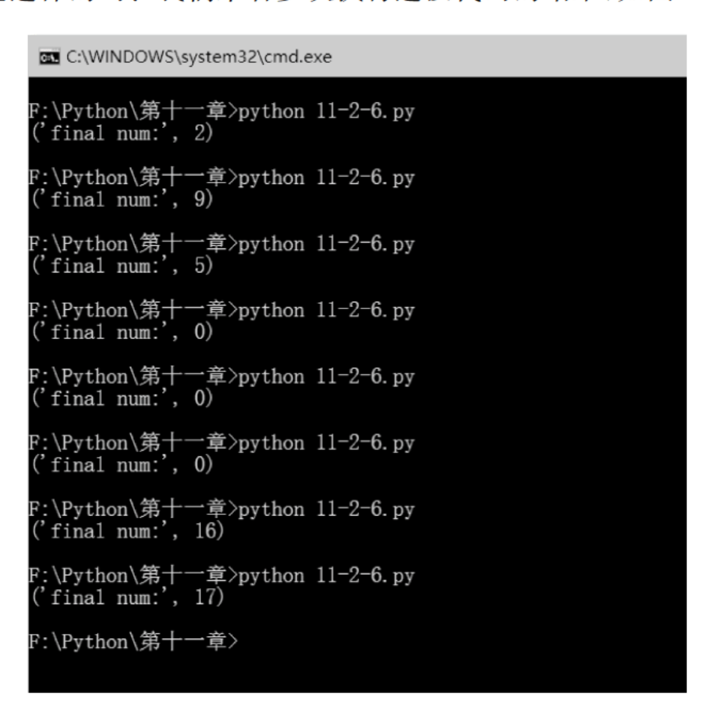


图 11.7 不安全的线程

从图 11.7 中可以看到，我们每次执行这段代码，最终输出的 num 并不总是 0，反而还有一些莫名其妙的值如 2、9、5、16、17。这些不应该出现的值是怎样产生的呢？答案就是线程不安全。

在之前介绍 Pyhon 的 CPython 解释器时，我们说到 CPyhon 引入了 GIL 使 CPython 解释器同时只能运行一个 bytecode，那么，我们就从 bytecode 的视角分析这段代码究竟发生

了什么。我们使用 dis 模块，输出 decNum 函数所对应的 bytecode（如果读者读不懂 bytecode 也没有关系，笔者将阐释每条 bytecode 的功能），在 num -= 1 行，我们得到如下几条 bytecode：

```
1 LOAD_GLOBAL              2 (num)
2 LOAD_CONST               1 (1)
3 INPLACE_SUBTRACT
4 STORE_GLOBAL             2 (num)
5 LOAD_CONST               0 (None)
6 RETURN_VALUE
```

这段 bytecode 清晰地揭示了 Python 在执行 num -= 1 时的工作原理：首先，第 1 行将全局变量 num 压入栈（这里的栈指程序的堆栈），第 2 行再向栈中压入常量 1，第 3 行将栈中两个值做减法，第 4 行将得到的结果存储到全局变量 num 中，5、6 两行用作函数返回，因为 decNum 无返回值，所以其返回了 None，无须关注。

从这段 bytecode 中我们可以看到，即使是 num -= 1 这条简单的语句，在 Python 运行时也不是被一条 bytecode 执行完的。因此，在多线程中，可能会出现这种情况：num 初始为 100，线程 1 将 num 当前值压入属于其自身的栈帧中（不同栈帧由 Python 的运行时控制，不会互相影响）后还没来得及进行后续的减法操作，CPython 就切换到了下一线程，而在线程 2 中，由于 num 还未进行减法运算或者之前线程减法运算的结果还未存储到 num 中，所以线程 2 取得的 num 值仍为 100，当这两个线程都结束后（暂时不考虑更多线程），num 的值都被存储为 100－1 即 99 而非 98。这便导致了“线程安全问题”。GIL 保证的线程安全为 bytecode 的线程安全，而非所有操作的线程安全，因此在使用多线程操作数据时，我们仍需对共享的数据加锁。

由此可见，如果多个线程没有操作同一对象，是不会出现线程安全问题的。然而，真实的情况是我们往往需要在多个线程中访问同一对象，那么，我们应该如何访问才不会出现难以察觉的线程安全问题呢？这就是线程锁诞生的原因。

线程锁即用于线程间访问控制的锁，当一个线程使用一线程锁加锁时，其他线程无法请求该线程锁，这些线程将会被阻塞，只有该线程锁被释放时，被阻塞的线程才能继续运行。

Python 提供了多种线程锁来方便我们进行多线程开发，接下来将介绍常用的线程锁。

5. 互斥锁

互斥锁是 threading 模块提供的最简单的线程锁，因为这种线程锁使用 acquire()加锁、使用 release()对解锁，所以被称为互斥锁。

互斥锁的使用非常简单，我们可以在全局使用 threading.Lock()实例化一个互斥锁，在子线程操作共享的对象前使用 acquire()加锁，操作完成后使用 release()解锁：

```
import threading
import time


def decNum():
    global num
```

```
        time.sleep(1)
        mutex.acquire()
        num -= 1
        mutex.release()


num = 100
thread_list = []

mutex = threading.Lock()

for i in range(100):
    t = threading.Thread(target=decNum)
    t.start()
    thread_list.append(t)

for t in thread_list:
    t.join()

print('final num:', num)
```

加入互斥锁后的执行效果如图 11.8 所示。



```
C:\WINDOWS\system32\cmd.exe

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>python 11-2-7.py
('final num:', 0)

F:\Python\第十一章>
```

图 11.8　使用互斥锁后的结果

正如之前所说，其他线程在试图操作请求线程锁 mutex 时会被阻塞，直到其解锁才能继续操作，因此 num -= 1 操作不会因切换线程而出现线程不安全问题。

其中互斥锁的 acquire()函数有可选的 blocking 参数，默认为 True，允许阻塞线程，当 acquire(False)时，如果请求加锁失败则会直接返回 False，请求成功则会返回 True：

```
import threading
import time

def func_1():
    mutex.acquire()
    time.sleep(10)
    mutex.release()

def func_2():
    time.sleep(1)
    print "Try to get Lock"
    print mutex.acquire(False)

mutex = threading.Lock()

t1 = threading.Thread(target = func_1)
t2 = threading.Thread(target = func_2)

t1.start()
t2.start()
```

这段代码的运行效果如图 11.9 所示。



图 11.9　关闭阻塞功能

如图 11.9 所示，t1 线程执行时加锁并阻塞 10 秒，t2 再执行 1 秒后使用 acquire(False)不阻塞地请求锁，显然此时 t1 的锁还未释放，函数返回 False。

虽然互斥锁可以解决多线程操作同一资源时产生的线程安全问题，然而有时，互斥锁会导致另一种更难发现的线程安全问题的产生：死锁。

死锁不是一种线程锁，而是错误地使用线程锁而导致的另一种线程安全问题。例如，当我们交叉请求锁时，即一个线程先请求 A 锁再请求 B 锁后释放 B 锁再释放 A 锁，另一线程先请求 B 锁再请求 A 锁后释放 A 锁再释放 B 锁，这两个线程并发时，如果线程 1 锁定了 A 锁后切换到线程 2 锁定了 B 锁，此时，线程 1 无法再请求 B 锁，线程 2 也无法请求 A 锁，形成交叉死锁。例如下面这段代码演示了交叉死锁的形成：

```
import threading
import time


class MyThread(threading.Thread):
```

```
    def __init__(self, id):
        threading.Thread.__init__(self)
        self.id = id
        pass

    def do1(self):
        if mutexA.acquire():
            print str(self.id) + ":Get A !"
            if mutexB.acquire():
                print str(self.id) + ":Get B !"
                mutexB.release()
                print str(self.id) + ":Release B !"
            mutexA.release()
            print str(self.id) + ":Release A !"

    def do2(self):
        if mutexB.acquire():
            print str(self.id) + ":Get B !"
            if mutexA.acquire():
                print str(self.id) + ":Get A !"
                mutexA.release()
                print str(self.id) + ":Release A !"
            mutexB.release()
            print str(self.id) + ":Release B !"

    def run(self):
        self.do1()
        self.do2()

mutexA = threading.Lock()
mutexB = threading.Lock()


def test():
    for i in range(10):
        t = MyThread(i)
        t.start()

if __name__ == '__main__':
    test()
```

当我们运行以上代码时，得到如图 11.10 所示的结果(不同情况下结果可能不同，且由于 Python 2.x 的 print 函数非线程安全，输出可能会部分混乱)。

可以看到，在这段代码中，本应由 10 个线程交叉请求 AB 锁，然而线程 0、1 已经形成死锁，程序不会继续运行。从输出中分析，线程 0 执行完 do1 后，线程 1 开始执行 do1，在线程 1 请求成功 A 锁后，线程 0 开始执行 do2 并请求成功 B 锁。此时，线程 1 在 do1 函数中需要请求 B 锁而线程 0 在 do2 函数中需要请求 A 锁，而两锁均为锁定状态，因此这两个线程形成了死锁，其他线程同样在阻塞请求 A 锁，程序无法进行。

```
python 11-2-9.py

F:\Python\第十一章>python 11-2-9.py
0:Get A !
0:Get B !
0:Release B !
0:Release A !1:Get A !

0:Get B !
```

图 11.10　死锁

其实 Python 的互斥锁迭代加锁也会产生死锁：

```
import threading
import time

def func():
    print "Start !"
    mutex.acquire()
    mutex.acquire()
    mutex.release()
    mutex.release()
    print "Finish !"

mutex = threading.Lock()

t = threading.Thread(target = func)

t.start()
```

这段代码的运行效果如图 11.11 所示。

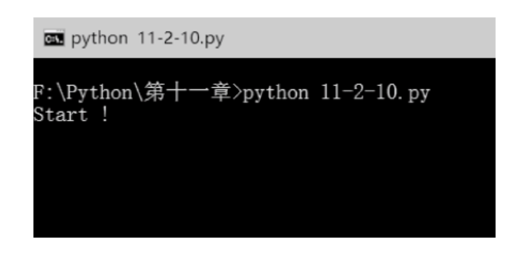


图 11.11　迭代互斥锁死锁

mutex 自身迭代加锁产生了死锁，线程 t 永远不会结束。

死锁一般是在调试时难以察觉的 bug，在编程时要极力避免产生死锁。解决死锁问题有大量的不同环境下的解决方案并需要大量的编程经验，已经超出本书的讨论范围。想要进一步了解死锁解决方案的读者可以在互联网中查询相关资料。

而对于自身迭代产生的死锁，threading 模块提供了一种递归锁来解决这一问题。

6. 递归锁

递归锁的使用方法与互斥锁基本相同，使用 threading.RLock()实例化递归锁，并使用

acquire()、release()请求、释放锁。与互斥锁不同的是，RLock 允许在同一线程中多次请求锁而不会产生死锁问题。使用 RLock 时必须严格使执行的 acquire()与 release()成对出现。

我们用递归锁替换互斥锁迭代使用：

```
import threading
import time

def func():
    print "Start !"
    rlock.acquire()
    rlock.acquire()
    rlock.release()
    rlock.release()
    print "Finish !"

rlock = threading.RLock()

t = threading.Thread(target = func)

t.start()
```

这段代码的运行效果如图 11.12 所示。

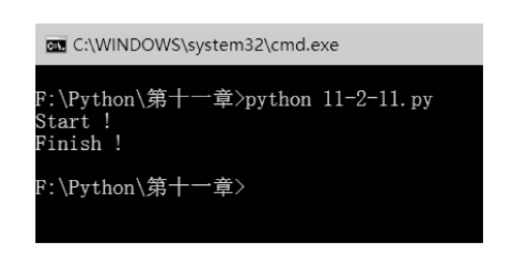


图 11.12　使用递归锁

线程 t 这次可以正常解锁并退出。

7. 使用 Condition 同步线程

除了线程锁，threading 模块还提供了状态类 Condition 来实现复杂的线程同步问题。使用 Condition 类时，我们同样需要创建 Condition 实例。Condition 实例除了具有互斥锁的 acquire()与 release()方法实现加锁、释放锁外，还有 wait()与 notify()等方法。

Condition 实例对于加解锁操作会维护一个锁定池。

wait()方法的作用是将已加锁的线程解锁并放入等待池并阻塞，等待其他线程的通知才能运行。这个方法只能对加锁的线程使用，否则会抛出异常。

notify()方法则是从等待池中取出一个线程并通知，被通知的线程将自动调用 acquire()方法尝试获得锁定。这个方法有一点很重要，它不会释放线程的锁定，使用前线程必须已获得锁定，否则将抛出异常。

在本节的例子中，我们将使用 Condition 实现双线程的简化版生产消费者模型，我们分

别创建生产者线程与消费者线程。常见的生产消费者模型有如下特点：

(1) 生产者仅仅在仓储未满的时候生产，仓满则停止生产。

(2) 消费者仅仅在仓储有产品的时候才能消费，仓空则等待。

(3) 当消费者发现仓储没产品可消费的时候会通知生产者生产。

(4) 生产者在生产出可消费产品的时候，应该通知等待的消费者去消费。

而我们要实现的简化版模型不考虑仓储容量，即有仓储容量为1，有产品则仓满，否则仓储为空。简单来说，这个简化版模型就像是两个人在对话，每人在对方说话后只能说一句话。通过Conditon类，我们可以便捷地实现：

```
import threading
import time

product = None

con = threading.Condition()


def produce():
    global product
    while True:
        con.acquire()
        if product is not None:
            con.wait()
        print 'Prodecing...'
        time.sleep(2)
        product = '***Product***'
        con.notify()
        con.release()



def consume():
    global product
    while True:
        con.acquire()
        if product is None:
            con.wait()
        print 'Consuming...'
        time.sleep(2)
        product = None
        con.notify()
        con.release()


t1 = threading.Thread(target=produce)
t2 = threading.Thread(target=consume)

t1.start()
t2.start()
```

这段代码的运行效果如图 11.13 所示。

```
python 11-2-12.py

F:\Python\第十一章>python 11-2-12.py
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
Prodecing...
Consuming...
```

图 11.13 使用 condition 同步线程

从输出可见,生产和消费活动在两个线程中实现了交替进行(且无论先启动生产者线程还是消费者线程,总是先生产再消费)。我们简单分析一下代码,生产者与消费者的逻辑类似。生产者、消费者都会首先请求锁,请求成功后使用循环检测全局 product(产品,本例中即仓储),当仓储为空或满时,生产或消费,之后使用 notify()方法通知另一线程生产或消费完成;无论仓空或是仓满,二者都会随后使用 wait()方法阻塞本线程并等待对方线程发出通知,当线程恢复运行状态后,为了观察清晰,我们让线程等待 2 秒运行。

从这个例子中我们发现 Condition 可以便捷地实现一些复杂的线程同步问题。

8. 使用 Event 实现线程间通信

Event 类其实可以看作简化版的 Condition 类,它也能阻塞线程等待信号、发出信号恢复阻塞中的线程,但是 Event 类不提供线程锁的功能。

使用 Event 前同样需要实例化 Event 对象,Event 实例内部维护一个布尔变量,表示线程运行的状态。

- isSet()方法用来返回内部的布尔变量值。
- wait()方法将使该线程阻塞,直到其他线程调用 set()方法。
- set()方法将布尔变量置为 True,并通知所有阻塞中的线程恢复运行。
- clear()方法会将内部布尔变量置为 False。

我们使用 Condition 类实现上例中简化版的生产消费者模型:

```
import threading
import time

product = None

event = threading.Event()
```

```
def produce():
    global product
    event.set()
    while True:
        if product is None:
            print 'Prodecing...'
            product = '***Product***'
            event.set()
        event.wait()
        time.sleep(2)


def consume():
    global product
    event.wait()
    while True:
        if product is not None:
            print 'Consuming...'
            product = None
            event.set()
        event.wait()
        time.sleep(2)

t1 = threading.Thread(target = produce)
t2 = threading.Thread(target = consume)


t2.start()
t1.start()
```

同样，我们得到与上例相同的输出，如图 11.14 所示。

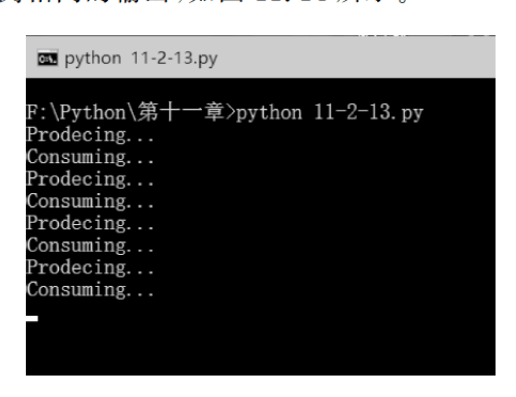


图 11.14　使用 Event 实现线程间通信

9. 使用 Timer 定时器

Timer 类实际上是 Thread 类派生出的一个类。使用 Timer 一样需要实例化 Timer 对象，实例化时有三个重要的参数需要传递，第一个是启动延迟时间，第二个是启动调用的函数，第三个是函数参数(可以不设置)；实例化之后，调用 start()方法即可启动定时器。

```
import threading


def func():
    print "I Love Python !"
    print "Now is : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.
time()))

timer = threading.Timer(2,func)

print "Now is : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time
()))
timer.start()
```

这段代码的运行效果如图 11.15 所示。

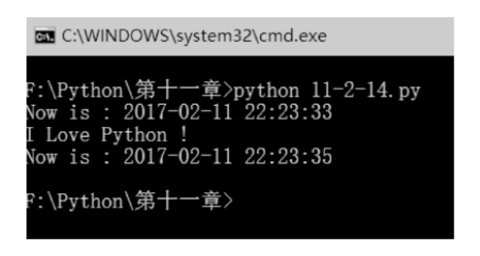


图 11.15　使用 Timer 定时器

10. 使用 local 线程局部字典

为了方便地存储不同线程的数据，threading 模块还提供了 local 类。local 类可以为不同线程存储互不干扰的同名数据。在使用时，需要在主线程实例化 local 类，然后便可以动态地追加、修改它的属性，而且这些属性在不同线程之间即使同名也不互相干扰：

```
import threading
import time
import random

localVal = threading.local()
localVal.val = "Thread-Main"


def func(val):
    localVal.val = val
    time.sleep(random.random() * 2)
    print "%s == %s" % (val, localVal.val)
    print localVal.__dict__


t1 = threading.Thread(target=func, args=("Thread-1",))
```

```
t2 = threading.Thread(target = func, args = ("Thread - 2",))

t1.start()
t2.start()
t1.join()
t2.join()

print "%s == %s" % ("Thread - Main", localVal.val)
print localVal.__dict__
```

这段代码的运行效果如图 11.16 所示。



图 11.16　使用 local 局部字典

可见，我们在主线程与两个子线程均修改了 local 实例的 val 值，而输出的 val 只能输出当前线程赋给 local 实例的值。local 类更像是一个字典，我们可以使用__dict__属性以列表形式返回当前线程下为 local 实例赋予的属性与对应的值。

11.3　Python 多进程编程

11.3.1　Python 多进程编程的特点

在介绍 Python 多线程编程时，我们了解到由于一些 Python 解释器的原因，很多时候 Python 并不能让多个线程真正地并行，降低了多核 CPU 的利用率。而 Python 多进程运行时，每个进程有自己独立的 GIL，可以充分利用多核 CPU 的性能。

Python 同样为多进程编程提供了 multiprocessing 模块，使开发者可以便捷地开发多进程的 Python 程序。

11.3.2　使用 multiprocessing 模块进行多进程编程

Python 的自带模块 multiprocessing 提供了便捷的多进程开发功能，使用前先导入模块：import multiprocessing。multiprocessing 模块提供的类使用方法与多线程编程时使用的 threading 模块很相似。

1. 创建与运行进程

multiprocessing 模块的使用方式与 threading 很相似，在创建新进程时，同样有直接使

用 multiprocessing. Process()返回新进程对象与使用自定义类继承 multiprocessing. Process 两种方式。不过需要注意的是,在 Windows 下使用 multiprocessing 编写多进程程序时,要在主进程开始前使用 if __name__ == "__main__":判断当前进程是否为主进程,并调用 multiprocessing. freeze_support()函数,使代码只有为主进程时才能执行,否则因为在 Windows 下 Python 子进程会自动 import 主进程中的内容,导致未判断的主进程中代码在子进程中被死递归调用而出错。

Process 与 Thread 相似,具有 target 与 args 参数,分别为该进程执行的函数与其对应的参数。实例化 Process 对象后,同样需要调用实例方法 start()启动进程。

这段代码展示了直接使用 multiprocessing. Process()返回新进程对象的方法创建新进程:

```
import multiprocessing
import time

def func(id):
    print "Process %d started !\n" % id
    if(id == 1):
        time.sleep(2)
    print "Process %d finished !\n" % id

if __name__ == "__main__":

    multiprocessing.freeze_support()

    p1 = multiprocessing.Process(target=func, args=(1,))
    p2 = multiprocessing.Process(target=func, args=(2,))

    p1.start()
    p2.start()
```

这段代码展示了如何使用自定义类继承 multiprocessing. Process 创建新进程:

```
import multiprocessing
import time


class MyProcess(multiprocessing.Process):

    def __init__(self, id):
        multiprocessing.Process.__init__(self)
        self.id = id

    def run(self):
        print "Process %d started !\n" % self.id
        if(self.id == 1):
```

```
            time.sleep(2)
        print "Process %d finished !\n" % self.id


if __name__ == "__main__":

    multiprocessing.freeze_support()

    p1 = MyProcess(1)
    p2 = MyProcess(2)

    p1.start()
    p2.start()
```

这两段代码的运行效果相同,如图 11.17 所示。

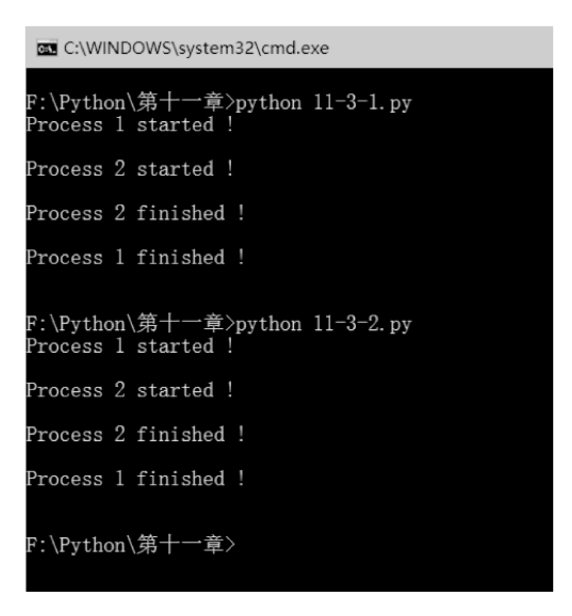


图 11.17　创建并运行多进程

2. 使用 join()函数阻塞进程

多进程编程中,同样可以对子进程使用 join()方法来阻塞当前进程,使被 join()的进程全部执行完毕后继续执行当前进程:

```
import multiprocessing
import time


def func(sleeptime):
    time.sleep(sleeptime)
    print "Process which slept %d second finished !\n" % sleeptime

p1 = multiprocessing.Process(target = func, args = (1,))
```

```
p2 = multiprocessing.Process(target = func, args = (2,))

if __name__ == "__main__":

    multiprocessing.freeze_support()

    print "All Processes start at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time())) + "\n"

    p1.start()
    p2.start()

    p1.join()

    print "Now is : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.
time())) + "\n"
```

这段代码的运行效果如图 11.18 所示。

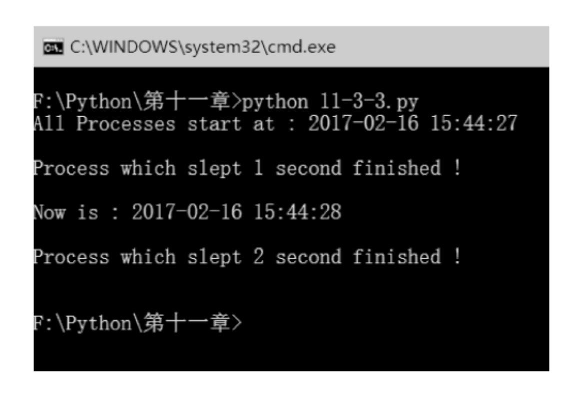


图 11.18　使用 join 阻塞进程

另外,join()方法的超时用法与多线程的 join()方法相同。

3. 守护进程

当主进程结束时,守护进程随之结束而非守护进程依旧继续执行。将进程设置为守护进程的方式与设置守护线程略有不同,设置守护进程需要将 Process 的实例的 deamon 属性修改为 True:

```
import multiprocessing
import time


def func():
    print "Process started !"
    time.sleep(5)
    print "Process finished !"
```

```
if __name__ == "__main__":

    multiprocessing.freeze_support()

    print "Main Process started at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time()))

    p = multiprocessing.Process(target = func)
    p.daemon = True
    p.start()

    time.sleep(2)
    print "Main Process finished at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time()))
```

这段代码的运行效果如图 11.19 所示。

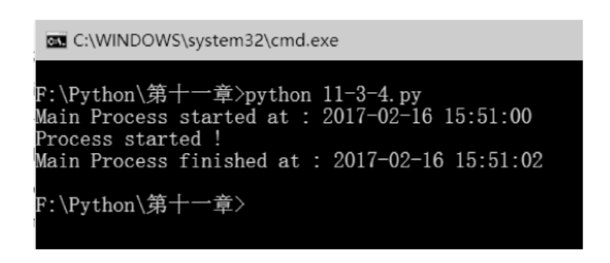


图 11.19　守护进程

可见，主进程结束后，守护进程直接被杀死，没有继续执行。

4. 进程锁

与多线程安全问题类似，多进程程序同样有进程安全问题，例如在多进程读写文件时，如果处理方式不当就会产生进程安全问题。

为了解决进程安全问题，multiprocessing 模块也提供了 Lock（互斥锁）与 RLock（递归锁）类，其用法与线程锁类似：

```
import multiprocessing

def worker_with(lock, f, text):
    lock.acquire()
    with open(f, "a+") as fs:
        fs.write(text + '\n')
    lock.release()

if __name__ == "__main__":

    multiprocessing.freeze_support()

    f = "test.txt"
```

```
        lock = multiprocessing.Lock()

        for i in range(10):
            multiprocessing.Process(target = worker_with, args = (lock, f, 'No.' + str(i))).
start()
```

这段代码运行后得到 test.txt 内容如下：

```
No.2
No.3
No.1
No.0
No.6
No.7
No.5
No.8
No.4
No.9
```

可见，0～9 号都被写入了文件中（根据执行状况不同，序号顺序会不同），我们再看不使用进程锁的情况：

```
No.2
No.4
No.0
No.6
No.7
No.5
No.9
No.8
```

可见，不使用进程锁，由于进程安全问题，有些序号没有被呈现到最终的文件中。

5. 使用 Semaphore 类控制资源同时访问数量

有些情况下，我们需要限制同时执行的最大进程数，这时我们可以使用 Semaphore 类。Semaphore 类实例化时可以设置最大同时访问资源的进程数，使用 Semaphore 的 acquire() 方法与 release() 方法来请求与释放 Semaphore 控制，只有当同时获取 Semaphore 的进程数小于设定的最大值时，请求才会成功，否则会阻塞到有一进程释放：

```
import multiprocessing
import time

def worker(s, i):
    s.acquire()
    print(multiprocessing.current_process().name + "acquire");
    time.sleep(i)
```

```
        print(multiprocessing.current_process().name + "release");
        s.release()

if __name__ == "__main__":

        multiprocessing.freeze_support()

        s = multiprocessing.Semaphore(2)

        for i in range(5):
            p = multiprocessing.Process(target = worker, args = (s, i + 1))
            p.start()
```

这段代码的运行效果如图 11.20 所示。



图 11.20　使用 Semaphore 限制资源访问数

从输出中我们可以观察到，同时持有 Semaphore 的进程数最多仅为 2。

6. 使用 Value 与 Array 类在进程间共享变量

多进程 Python 程序运行时采用多 GIL 并行，因此多进程程序不能使用 global 关键字访问全局变量。为此，multiprocessing 模块提供了 Value 类与 Array 类使变量可以在内存中共享(使用时，请注意进程安全问题)。Value 类型和 Array 类型实例化时均需要两个参数，一个是共享变量的类型，另一个是共享变量的值。其中类型需要使用 Type code 表示，详见表 11.1。

表 11.1　Type code

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2

续表

Type code	C Type	Python Type	Minimum size in bytes
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	4

需要注意的是，Value 与 Array 类自身同样不会保证进程安全，需要使用进程锁或者 multiprocessing 模块下的 Condition 类或 Event 类通信保证进程安全(multiprocess 模块下的 Conditon 类与 Event 类与多线程中用法类似，不再赘述)。

例如，我们使用 Value 与 Event 类实现生产消费者模型(本例 Conditon 使用的方式与多线程中 Condition 使用的方式稍有不同，两种方式都可以实现，本例中请求锁后一直占有，使用 wait()与 notify()方法，而多线程一节中则除了使用 wait()与 notify()外还不断地释放、重新请求锁，相比下，多线程一节中 Condition 的使用方式更加规范)：

```
import multiprocessing
import time


def produce(event, v):
    event.set()
    while True:
        if v.value == 'x':
            print 'Prodecing...'
            v.value = 'o'
            event.set()
        event.wait()
        time.sleep(2)


def consume(event, v):
    event.wait()
    while True:
        if v.value == 'o':
            print 'Consuming...'
            v.value = 'x'
            event.set()
        event.wait()
        time.sleep(2)


if __name__ == "__main__":

    multiprocessing.freeze_support()
```

```
    product = multiprocessing.Value('c', 'x')

    event = multiprocessing.Event()

    p1 = multiprocessing.Process(target = produce, args = (event, product))
    p2 = multiprocessing.Process(target = consume, args = (event, product))

    p2.start()
    p1.start()
```

这段代码的运行效果如图 11.21 所示。

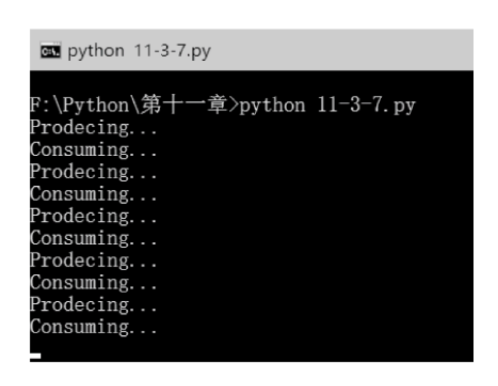


图 11.21　多进程生产消费模型

7. 使用 Pipe 在两进程间通信

有时我们需要将一个进程中的一些值传递给另一进程使用，这时，我们可以使用 multiprocessing 模块提供的 Pipe 类。

Pipe 类实例化时会返回管道的两端，默认情况下两端可以互相通信，如果实例化时使用 False 参数则只允许第一个管道端给第二个管道端发送信息。管道端有 send()与 recv()方法，在管道一端 send()的内容可以在另一端通过 recv()获取：

```
import multiprocessing
import time

def sender(pipe):
    pipe.send("I love Python !")

def recver(pipe):
    time.sleep(2)
    print pipe.recv()

if __name__ == "__main__":

    multiprocessing.freeze_support()

    (pipe_1,pipe_2) = multiprocessing.Pipe()
```

```
        p1 = multiprocessing.Process(target = sender,args = (pipe_1,))
        p2 = multiprocessing.Process(target = recver,args = (pipe_2,))

        p1.start()
        p2.start()
```

这段代码的运行效果如图 11.22 所示。

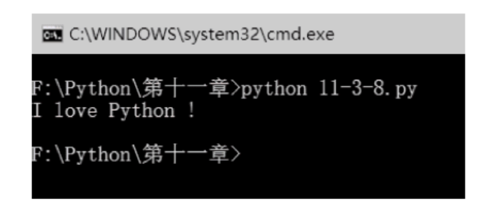


图 11.22　使用 Pipe 的多进程通信

8. 使用 Queue 实现多进程通信

Pipe 类可以允许两进程间通信，而有时我们需要更多进程间通信。例如在并行计算时，我们可能需要将大规模的数据并行计算，可能一次将所有要运算的数据分配给进程所需进程数不足，所以我们需要一些进程运算完成后再取剩下的数据运算，此时，我们可以使用 multiprocessing 模块下的 Queue 类实现。

顾名思义，Queue 类是一个队列类，multiprocessing 模块下的 Queue 类操作与一般的 Queue 类相仿，而 multiprocessing 模块下的 Queue 类支持多进程共享。实例化 Queue 类时需要一个整型参数作为队列的大小。

multiprocessing 模块下的 Queue 实例有表 11.2 所示的常用方法。

表 11.2　Queue 常用方法

Queue. qsize()	返回队列的实际大小
Queue. empty()	如果队列为空，返回 True，反之返回 False
Queue. full()	如果队列已满，返回 True，反之返回 False
Queue. put(item[,timeout])	向队尾添加，timeout 等待时间(队满时会阻塞)
Queue. get([block[, timeout]])	获取队列，timeout 等待时间(队空时会阻塞)
Queue. put_nowait(item)	非阻塞 put，失败时会抛出异常，相当于 Queue. put(item, False)
Queue. get_nowait(item)	非阻塞 get，失败时会抛出异常，相当于 Queue. get(False)

例如我们使用 Queue 完成多生产者单消费者的程序：

```
import multiprocessing
import time


def produce(queue, id):
    timer = 0
    while True:
```

```
            time.sleep(1)
            timer += 1
            queue.put('*** Process:' + str(id) + '- Product - ' + str(timer) + '***')


def consume(queue):
    while True:
        print queue.get()


if __name__ == "__main__":

    multiprocessing.freeze_support()

    queue = multiprocessing.Queue(10)
    for i in range(3):
        multiprocessing.Process(target = produce, args = (queue, i)).start()
    multiprocessing.Process(target = consume, args = (queue, )).start()
```

这段代码的运行效果如图 11.23 所示。



图 11.23　使用 Queue 的多进程通信

9. Pool 进程池

对于少量进程并行，我们只需要创建进程执行即可。而有时我们可能需要创建数十个甚至上百个进程，此时，我们需要设置进程最大同时执行数来平衡线程的消耗。之前介绍的 Semaphore 可以实现这一功能，然而它采用的是类似锁的形式，操作比较烦琐，适用于对资源访问数量的控制。为了方便对进程数进行控制，我们可以使用 multiprocessing 模块提供的 Pool 类。

实例化 Pool 对象时，需要设置 processes 参数，该参数表示允许同时运行的进程数。实例化 Pool 对象后，我们只需要操作 Pool 的实例即可。

Pool 的实例有如表 11.3 所示的常用的方法。

表 11.3　Pool 的常用方法

方　法	说　明
apply_async(func[, args[, kwds[, callback]]])	添加异步进程，其他进程不会等待其执行后执行(主进程结束时，异步进程会立即结束)
apply(func[, args[, kwds]])	添加同步进程，其他进程会等待其执行后执行
close()	封锁线程池，使其不再接受新的任务
terminate()	关闭线程池中的所有任务
join()	阻塞主线程，等待 pool 中所有任务执行后再执行，使用 join 方法可用于使主进程等待异步进程结束后结束(join 方法只能在 close()或 terminate()后使用)

观察本例体会异步进程与同步进程的区别：

```
import multiprocessing
import time


def func(sleeptime, id):
    time.sleep(sleeptime)
    print "Process-" + str(id) + " finished at : " + time.strftime('%Y-%m-%d %H:
%M:%S', time.localtime(time.time())) + "\n"

if __name__ == '__main__':

    multiprocessing.freeze_support()

    pool = multiprocessing.Pool(processes=3)

    pool.apply(func, (2, 1))
    pool.apply(func, (2, 2))
    pool.apply_async(func, (1, 3))
    pool.apply_async(func, (10, 4))

    print "Main Process ran at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(time.time())) + "\n"
    time.sleep(5)
```

这段代码的运行效果如图 11.24 所示。

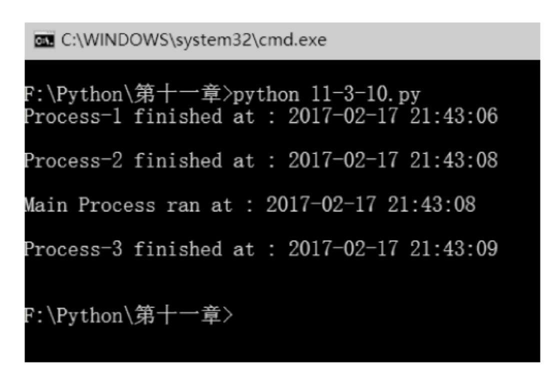


图 11.24　使用 Pool 进程池

从输出中我们可以看出，同步进程1、2执行时其他进程处于阻塞状态，等其执行完后，主进程与异步进程3同时开始执行，异步进程3等待1秒后输出，而异步进程4在主进程结束前仍未输出，因此它与主进程一同结束，无输出。

为了使异步进程执行完毕再结束主进程，我们使用join()方法阻塞主进程：

```
import multiprocessing
import time


def func(sleeptime, id):
    time.sleep(sleeptime)
    print "Process-" + str(id) + " finished at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())) + "\n"

if __name__ == '__main__':

    multiprocessing.freeze_support()

    pool = multiprocessing.Pool(processes=3)

    pool.apply(func, (2, 1))
    pool.apply(func, (2, 2))
    pool.apply_async(func, (1, 3))
    pool.apply_async(func, (10, 4))

    print "Main Process ran at : " + time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())) + "\n"

    pool.close()
    pool.join()
```

这段代码的运行效果如图11.25所示。

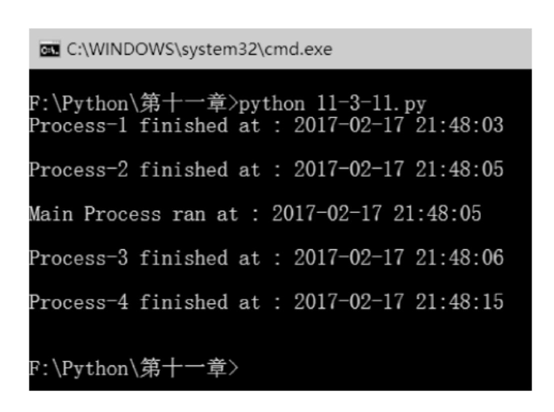


图11.25　加入join的进程池

可见，使用join后，主进程会等待pool中所有进程执行结束后再退出。

习 题 11

一、选择题

1. (　　)是计算机中的程序关于某数据集合上的一次运行活动,是系统进行资源分配和调度的基本单位,是操作系统结构的基础。

A. 线程　　B. 进程　　C. 资源　　D. 程序

2. (　　)是操作系统能够进行运算调度的最小单位(程序执行流的最小单元)。

A. 线程　　B. 进程　　C. 资源　　D. 程序

3. (　　)解决了互斥锁多次加锁后死锁的情况。

A. 死锁　　B. 互斥锁　　C. 递归锁　　D. 进程锁

二、填空题

1. ________是计算机中的程序关于某数据集合上的一次运行活动,是系统进行资源分配和调度的基本单位,是操作系统结构的基础。在早期面向进程设计的计算机结构中,进程是程序的基本执行实体。

2. ________是操作系统能够进行运算调度的最小单位(程序执行流的最小单元)。它被包含在进程之中,是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流,一个进程中可以并发多个线程,每条线程并行执行不同的任务。

3. 我们平时使用的 GUI 程序,________(是/否)多线程程序。

4. Python 为多线程编程提供了________模块,为多进程编程提供了________模块。

三、论述题

1. 如何使用多线程安全地读写一个文本文件,保证每次写入一行的内容连续。

2. 在 CPython 环境下,对于 CPU 密集的操作,我们应该使用多线程编程还是多进程编程?为什么?

四、编程题

使用多线程或多进程写入文本,每个进程写入一行由 100 个相同字符组成的字符串,同时有 100 个进程或线程并发,保证每行内容完整且正确。

第12章　Python 访问数据库

数据库(Database)是按照数据结构来组织、存储和管理数据的建立在计算机存储设备上的仓库。数据库中的数据以一定的数据模型组织、描述和存储在一起、具有尽可能小的冗余度、较高的数据独立性和易扩展性的特点并可在一定范围内为多个用户共享。

使用数据库,可以为多项服务共享数据,为系统编程提供便利,本章将通过介绍使用 Python 操作两种数据库与一种专为 Python 开发的数据库工具来讲解 Python 如何使用数据库。

12.1　使用 SQLite

12.1.1　SQLite 简介

SQLite 是一款轻型且遵守 ACID 的关系型数据库管理系统,是 D. RichardHipp 建立的公有领域项目。它的设计目标是嵌入式的,而且目前已经在很多嵌入式产品中使用了,它占用资源非常的少,在嵌入式设备中,可能只需要几百千字节的内存就够了。它能够支持 Windows/Linux/UNIX 等主流的操作系统,并且处理速度非常快。这些优势使 SQLite 成为现在的热门数据库之一。下面我们将介绍如何使用 python 自带模块 sqlite3 来操作 SQLite 数据库。



视频讲解

12.1.2　使用 sqlite3 模块操作 SQLite

1. 创建数据库

数据库大多使用于服务器,因此我们将使用 Ubuntu 16.04 系统演示数据库操作。SQLite 与 Python 均可以跨平台,因此在 Windows 下大同小异,只有在安装时略有不同。

如果读者的 Ubuntu 系统中没有安装 SQLite3,可输入命令 sudo apt-get install sqlite3 进行安装,如图 12.1 所示。Windows 系统可访问 SQLite 的官方下载地址 http://www.sqlite.org/download.html 下载 Windows Shell 版,解压后通过命令行使用 sqlite3.exe 即可,如图 12.2 所示。

首先,在用户～/目录下,输入 mkdir sqlite 命令,新建一个名为 sqlite 的文件夹,再输入 cd sqlite 命令进入文件夹。在 sqlite 文件夹目录下,输入 sqlite3 test.db 新建数据库。因为我们意在演示使用 Python 操作数据库,所以进入数据库后输入.quit 退出数据库即可,如图 12.3 所示。

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ sudo apt-get install sqlite3
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  libjpeg62 libpango1.0-0
Use 'sudo apt autoremove' to remove them.
Suggested packages:
  sqlite3-doc
The following NEW packages will be installed:
  sqlite3
0 upgraded, 1 newly installed, 0 to remove and 49 not upgraded.
Need to get 0 B/515 kB of archives.
After this operation, 1,938 kB of additional disk space will be used.
Selecting previously unselected package sqlite3.
(Reading database ... 237931 files and directories currently installed.)
Preparing to unpack .../sqlite3_3.11.0-1ubuntu1_amd64.deb ...
Unpacking sqlite3 (3.11.0-1ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up sqlite3 (3.11.0-1ubuntu1) ...
ubuntu@ubuntu:~$
```

图 12.1　Ubuntu 下安装 SQLite 3

```
sqlite3.exe test.db
F:\Python>sqlite3.exe test.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite>
```

图 12.2　Windows 下通过命令行使用 sqlite3.exe 并新建 test.db 数据库

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~$ mkdir sqlite
ubuntu@ubuntu:~$ cd sqlite/
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> .quit
ubuntu@ubuntu:~/sqlite$
```

图 12.3　进入 sqlite3 命令行交互

2. 链接数据库

使用 Python 的 sqlite3 模块操作数据库时，首先要导入 sqlite3 模块，然后通过 sqlite3.connect(“数据库名”)的方法来连接数据库并取得链接实例。

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print 'Succeed to connect to sqlite !'

conn.close()
```

这段代码的运行效果如图 12.4 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ vim 12-1-1.py
ubuntu@ubuntu:~/sqlite$ python 12-1-1.py
Succeed to connect to sqlite !
ubuntu@ubuntu:~/sqlite$
```

图 12.4　链接数据库

取得链接实例后，我们便可以使用链接实例的成员函数来操作数据库了。

3. 创建数据表

在创建数据表时，我们需要使用 SQL 中的 CREATE TABLE 语句：

```
CREATE TABLE 表名(列名 1 类型与描述, 列名 2 类型与描述, 列名 3 类型与描述,...);
```

在本例中，我们将建立一个用于统计学生成绩的成绩表：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print 'Succeed to connect to sqlite !'

conn.execute(
         '''
                  CREATE TABLE rank
                  (
                           id INTEGER PRIMARY KEY NOT NULL,
                           name TEXT NOT NULL,
                           class INTEGER NOT NULL,
                           math INTEGER,
                           cpp INTEGER,
                           network INTEGER,
                           average REAL
                  )

         '''
)

print "Succeed to creat a table !"

conn.close()
```

这段代码创建了一个名为 rank 的表，其中有 7 列，分别为 id、name、class、math、cpp、network、average；其中 id 为主键，主键用于作为行的唯一标识，每张表中只能存在一个主键，主键永远不能更新，所以一般选择对该行信息没有意义的列作为主键(被称为对用户没

有意义原则)，如果不声明主键，SQLite 会自动创建一个隐藏的自增列作为主键；id、name、class 列还被设置了 NOT NULL 标记，即在插入数据时，这三列的数据必须被填写，否则在插入时会报错。

我们在 SQLite 中使用.tables 来查看数据库中所有的表，如图 12.5 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ vim 12-1-2.py
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> .tables
sqlite>
sqlite> .quit
ubuntu@ubuntu:~/sqlite$ python 12-1-2.py
Succeed to connect to sqlite !
Succeed to creat a table !
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> .tables
rank
sqlite> .quit
ubuntu@ubuntu:~/sqlite$
```

图 12.5　创建数据表

在 SQLite 3 中，我们还可以使用 SELECT * FROM sqlite_master WHERE name="rank"；来查询刚刚创建的表 rank 的内部结构(Select、Where 等用法将在后面的章节中详细讲解)，执行效果如图 12.6 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> select * from sqlite_master where name="rank";
table|rank|rank|2|CREATE TABLE rank
                (
                        id INTEGER PRIMARY KEY NOT NULL,
                        name TEXT NOT NULL,
                        class INTEGER NOT NULL,
                        math INTEGER,
                        cpp INTEGER,
                        network INTEGER,
                        average REAL
                )
sqlite> .quit
ubuntu@ubuntu:~/sqlite$
```

图 12.6　查询数据表

4. SQLite 数据类型

SQLite 具有很强的灵活性，它不强制约束数据类型，即任何类型的数据都可以插入到任何类型的列中(INTEGER PRIMARY KEY 除外，它只接受 64 位整数)。虽然如此，任何数据还是会有一个 SQLite 的存储类。

SQLite 的存储类十分简洁，只有 5 种，如表 12.1 所示。

表 12.1　SQLite 的存储类

存　储　类	描　　述
NULL	NULL 值
INTEGER	带符号的整型，根据其大小，会以 1、2、3、4、6 或 8 字节存储

续表

存　储　类	描　　述
REAL	浮点值，8 字节 IEEE 浮点数
TEXT	文本字符串，使用数据库编码(UTF-8、UTF-16BE 或 UTF-16LE)存储
BLOB	blob 数据，根据输入决定存储类

SQLite 支持列的亲和类型。任何列仍然可以存储任何类型的数据，当数据插入时，该字段的数据将会优先采用亲和类型作为该值的存储方式。在建表时所指定的类型，并不会约束插入的数据类型，它的作用是指示期望的类型。也就是说，如果向一个整型的列中插入一个字符串时，SQLite 会尝试把这个字符串转换成一个整型值，如果可以转换，则会插入整型，否则将插入字符串。这种特性被称为类型或列亲和性(Type or Column Affinity)。

SQLite 支持以下 5 种亲和类型，如表 12.2 所示。

表 12.2　亲和类型

亲 和 类 型	描　　述
TEXT	数值型数据在被插入之前，需要先被转换为文本格式，之后再插入到目标字段中
NUMERIC	当文本数据被插入到亲和性为 NUMERIC 的字段中时，如果转换操作不会导致数据信息丢失以及完全可逆，那么 SQLite 就会将该文本数据转换为 INTEGER 或 REAL 类型的数据，如果转换失败，SQLite 仍会以 TEXT 方式存储该数据。对于 NULL 或 BLOB 类型的新数据，SQLite 将不做任何转换，直接以 NULL 或 BLOB 的方式存储该数据。需要额外说明的是，对于浮点格式的常量文本，如"30000.0"，如果该值可以转换为 INTEGER 同时又不会丢失数值信息，那么 SQLite 就会将其转换为 INTEGER 的存储方式
INTEGER	对于亲和类型为 INTEGER 的字段，其规则等同于 NUMERIC，唯一差别是在执行 CAST 表达式时
REAL	其规则基本等同于 NUMERIC，唯一的差别是不会将"30000.0"这样的文本数据转换为 INTEGER 存储方式
NONE	不做任何的转换，直接以该数据所属的数据类型进行存储

SQLite 同时具有强大的兼容性，在保证自身灵活性的同时，SQLite 可以将传统的数据库存储类型转化为它所对应的亲和类型，如表 12.3 所示。

表 12.3　亲和类型

传统数据类型	亲 和 类 型
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER

续表

传统数据类型	亲 和 类 型
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB no datatype specified	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

5. 插入数据 INSERT INTO

在创建数据表后，我们便可以向数据库中插入数据。插入数据时，我们使用 INSERT 语句：

```
INSERT INTO 数据表名 (列 1,列 2,列 3,...) VALUES(数据 1,数据 2,数据 3,...);
```

同样，也可以使用 Python 的 sqlite3 模块来完成。

使用 Python 的 sqlite3 进行数据库操作时，一定要注意参数的使用方式，错误的参数使用方式可能会导致 SQL 注入漏洞，攻击者可以巧妙地构造数据将自己想要执行的 SQL 语句注入到你的 SQL 指令中，得到其想要得到的任何数据。

例如，我们在插入数据时，要避免采用字符串拼接的方式：

```
conn.execute("INSERT INTO 表名 (列 1,列 2,...) VALUES ('%s','%s',...)" % a,b);
```

而要采用“?”占位符，将参数以值的形式插入：

```
conn.execute("INSERT INTO 表名 (列 1,列 2,...) VALUES (?,?)" , (a,b));
```

（我们将在 UNION 一节中演示错误的提交方式造成的攻击。）

在进行插入、查询、更新等操作后，要调用连接对象的 commit()方法，该方法提交当前的事务。如果未调用该方法，那么自上一次调用 commit()以来所做的任何操作对其他数据

库连接来说都是不可见的，当 commit()执行后，数据才会真正被从缓存写入数据库。

例如，我们从控制台获取参数插入到 rank 表中时：

```
#!/usr/bin/python

import sqlite3

usr_id = raw_input("Please input an id:\n")
usr_name = raw_input("Please input a name:\n")
usr_class = raw_input("Please input a class:\n")

conn = sqlite3.connect('test.db')

conn.execute("INSERT INTO rank (id, name, class) VALUES (?,?,?)",(usr_id, usr_name, usr_class))
conn.commit()

conn.close()
```

这段代码运行前后如图 12.7 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> select * from rank;
sqlite>
sqlite> .quit
ubuntu@ubuntu:~/sqlite$ python 12-1-3.py
Please input an id:
1
Please input a name:
Tom
Please input a class:
1
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> select * from rank;
1|Tom|1||||
sqlite> .quit
ubuntu@ubuntu:~/sqlite$
```

图 12.7　插入数据

6. 查询数据 SELECT

查询数据时，使用 SELECT 语句：

```
SELECT 列 1,列 2,... FROM 表;
```

如果想查询所有列的信息，可以使用“*”：

```
SELECT * FROM 表;
```

使用 Python 的 sqlite3 模块查询时，会返回一个 cursor 类型的游标对象，通过 cursor 可以访问查询结果（同样，为了避免 SQL 注入，如果存在变量，可使用参数的方式进行提

交）。cursor 对象的 fetchone 会以元组的方式返回一行的查询结果，再次使用 fetchone 可以返回下一行结果，直到再无新行时返回空对象；而 fetchall 则会以列表的方式返回每一行查询结果的元组。我们使用命令行的方式进行演示（事先插入了 4 条数据），如图 12.8 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> conn = sqlite3.connect("test.db")
>>> curs = conn.execute("SELECT * FROM rank")
>>> curs.fetchone()
(1, u'Tom', 1, 67, 89, 77, None)
>>> curs.fetchone()
(2, u'Kate', 1, 90, 67, 84, None)
>>> curs.fetchone()
(3, u'Mark', 2, 57, 99, 85, None)
>>> curs.fetchone()
(4, u'Bill', 2, 64, 70, 88, None)
>>> curs.fetchone()
>>>
>>> curs = conn.execute("SELECT * FROM rank")
>>> curs.fetchall()
[(1, u'Tom', 1, 67, 89, 77, None), (2, u'Kate', 1, 90, 67, 84, None),
(3, u'Mark', 2, 57, 99, 85, None), (4, u'Bill', 2, 64, 70, 88, None)]
>>> 
```

图 12.8　查询数据

7. WHERE 子句

仅用 SELECT 只能进行简单的查询，若需要更多的条件限制，我们便需要使用一些子句对查询进行限定，WHERE 子句就是其中之一。

WHERE 子句可以限定查询范围，通过在 WHERE 后加上限定条件，可以实现丰富的查询功能，例如我们要查询 Tom 的所有信息：

```
SELECT * FROM rank WHERE name = 'Tom';
```

在子句中，还可以嵌套 SELECT 语句，例如我们希望找到 cpp 成绩最高的学生：

```
SELECT name FROM rank WHERE cpp = (SELECT MAX(cpp) FROM rank);
```

这两条语句运行的效果如图 12.9 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> conn = sqlite3.connect("test.db")
>>>
>>> conn.execute("SELECT * FROM rank WHERE name = 'Tom'").fetchone()
(1, u'Tom', 1, 67, 89, 77, None)
>>>
>>> conn.execute("SELECT name FROM rank WHERE cpp=(SELECT MAX(cpp) FROM rank)").fetchone()
(u'Mark',)
>>>
```

图 12.9　使用 WHERE 子句

8. LIMIT 与 OFFSET 子句

LIMIT 子句与 OFFSET 子句分别用于限制查询返回的总条数与查询的起始条数偏移

量。例如在 rank 表中，我们想查询第二条到第四条信息：

```
SELECT * FROM rank LIMIT 3 OFFSET 1 ＃偏移 1 条，查询 3 条
```

运行效果如图 12.10 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> conn = sqlite3.connect("test.db")
>>>
>>> curs = conn.execute("SELECT * FROM rank LIMIT 3 OFFSET 1")
>>> for row in curs:
...     print row
...
(2, u'Kate', 1, 90, 67, 84, None)
(3, u'Mark', 2, 57, 99, 85, None)
(4, u'Bill', 2, 64, 70, 88, None)
>>>
```

图 12.10　使用 LIMIT 与 OFFSET 子句

9. ORDER BY 子句

ORDER BY 子句用于对查询的数据排序，用户可以附加 ASC 升序排序，也可以附加 DESC 降序排序。例如，我们要按照 cpp 成绩降序查询学生信息：

```
SELECT * FROM rank ORDER BY cpp DESC
```

运行效果如图 12.11 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> conn = sqlite3.connect("test.db")
>>>
>>> curs = conn.execute("SELECT * FROM rank ORDER BY cpp DESC")
>>>
>>> for row in curs:
...     print row
...
(3, u'Mark', 2, 57, 99, 85, None)
(1, u'Tom', 1, 67, 89, 77, None)
(4, u'Bill', 2, 64, 70, 88, None)
(2, u'Kate', 1, 90, 67, 84, None)
>>>
```

图 12.11　使用 ORDER BY 子句

四人按照 cpp 的成绩顺序 99、89、70、67 排序。

10. 更新数据 UPDATE

当数据库中的数据需要更新时，需要使用 UPDATE 语句，UPDATE 的基本用法如下：

```
UPDATE 表名 SET 列 = 新值 WHERE 限定
```

（当调用 execute 后，不要忘记调用 commit 提交更改。）

例如我们需要将 Tom 的数学分数修改为 100，如图 12.12 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> conn = sqlite3.connect("test.db")
>>>
>>> conn.execute("SELECT * FROM rank WHERE name='Tom'").fetchone()
(1, u'Tom', 1, 67, 89, 77, None)
>>>
>>> conn.execute("UPDATE rank SET math=100 WHERE name='Tom'")
<sqlite3.Cursor object at 0x7f20f233ace0>
>>> conn.commit()
>>>
>>> conn.execute("SELECT * FROM rank WHERE name='Tom'").fetchone()
(1, u'Tom', 1, 100, 89, 77, None)
>>>
```

图 12.12　更新数据

11. SQLite 约束

约束是在表的数据列上强制执行的规则。这些是用来限制可以插入到表中的数据类型。这确保了数据库中数据的准确性和可靠性。约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。表 12.4 中列出了 SQLite 中的常用约束。

表 12.4　SQLite 约束

约束类型	描　述
PRIMARY KEY	主键，表中行的唯一标识
NOT NULL	确保某列不能有 NULL 值
DEFAULT	当某列没有指定值时，为该列提供默认值
UNIQUE	确保某列中的所有值是不同的
CHECK	确保某列中的所有值满足一定条件

PRIMARY KEY 与 NOT NULL 在之前的例子中做过讲解，在此不再赘述。

DEFAULT 约束显而易见用来为没指定具体值的列提供一个默认值。

UNIQUE 约束可以对单列使用，也可以对多列复合使用。对单列使用时，列中每个值必须不同，否则在插入或更新时会报错；对多列复合使用时，只有当两行的所有具有同一 UNIQUE 约束的列中值相同时才会报错。例如，我们新建一个 user 表存放用户账号、密码、姓、名，其中“账号”列单列使用 UNIQUE 约束，而“姓”“名”列两列复合使用 UNIQUE 约束。当插入两条信息的“账号”相同时，插入时会报错。当插入的两条信息含有相同的“姓“或者”名”时，SQLite 不会报错；而当插入的两条信息的“姓”和“名”都相同时，SQLite 才会报错。在 ubuntu 的 sqlite3 工具的命令行测试如图 12.13 所示。

CHECK 约束可以使用子句进行限定，例如我们创建 info 表统计用户年龄，年龄输入必须大于 0，当插入或更新的数据不满足 age 大于 0 时，SQLite 会报错，如图 12.14 所示。

12. 联合查询 UNION、UNION ALL

UNION 与 UNION ALL 用来将多个 SELECT 查询的行合并，UNION 与 UNION

ALL 语句均要求每次查询的列数相同。UNION 会合并相同的行而 UNION ALL 不会。例如，我们分别使用 UNION、UNION ALL 列出之前建立的三个表中所有的名字，如图 12.15 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> CREATE TABLE user(
   ...> username TEXT NOT NULL UNIQUE,
   ...> password TEXT NOT NULL,
   ...> lastname TEXT NOT NULL,
   ...> firstname TEXT NOT NULL,
   ...> UNIQUE(lastname,firstname)
   ...> );
sqlite>
sqlite> INSERT INTO user(username,password,lastname,firstname)VALUES("Tom","123456","刘","德华");
sqlite> INSERT INTO user(username,password,lastname,firstname)VALUES("Tom","234567","吴","彦祖");
Error: UNIQUE constraint failed: user.username
sqlite> INSERT INTO user(username,password,lastname,firstname)VALUES("Bill","123456","刘","翔");
sqlite> INSERT INTO user(username,password,lastname,firstname)VALUES("Lisa","123456","马","德华");
sqlite> INSERT INTO user(username,password,lastname,firstname)VALUES("Kate","123456","刘","德华");
Error: UNIQUE constraint failed: user.lastname, user.firstname
sqlite>
```

图 12.13　UNIQUE 约束

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> CREATE TABLE info(
   ...> name TEXT NOT NULL UNIQUE,
   ...> age INTEGER NOT NULL CHECK (age>0)
   ...> );
sqlite> INSERT INTO info (name,age)VALUES("Tom",15);
sqlite> INSERT INTO info (name,age)VALUES("Bill",-1);
Error: CHECK constraint failed: info
sqlite>
```

图 12.14　CHECK 约束

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT name FROM rank
   ...> UNION
   ...> SELECT username FROM user
   ...> UNION
   ...> SELECT name FROM info;
Bill
Kate
Lisa
Mark
Tom
sqlite> SELECT name FROM rank
   ...> UNION ALL
   ...> SELECT username FROM user
   ...> UNION ALL
   ...> SELECT name FROM info;
Tom
Kate
Mark
Bill
Bill
Lisa
Tom
Tom
sqlite>
```

图 12.15　UNION 联合查询

从图 12.15 中可以看出，UNION 合并了相同的行而 UNION ALL 则不会合并。

在学习 SQLite 之初，笔者便强调过在 Python 中使用 sqlite3 模块查询数据库时，变量要以参数方式传递而非字符串链接，字符串链接会导致 SQL 注入漏洞，泄露高危敏感信息，下面我们来对比两种方式：

```
#!/usr/bin/python
# coding = utf - 8

import sqlite3

mode = input("请选择模式:\n0:字符串链接 1:参数\n")
name = raw_input("请输入需要查询的学生名字\n")

conn = sqlite3.connect('test.db')

if(mode == 0):
    print conn.execute("SELECT math,cpp,network FROM rank WHERE name = '%s'" % name).
fetchall()
else:
    print conn.execute("SELECT math,cpp,network FROM rank WHERE name = ?" , (name,)).
fetchall()

conn.close()
```

在运行时，我们尝试构造 SQL 查询语句去闭合原语句，查询过程与结果如图 12.16 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-1-4.py
请选择模式：
0：字符串链接 1：参数
1
请输入需要查询的学生名字
Tom UNION SELECT username,username,password FROM user
[]
ubuntu@ubuntu:~/sqlite$ python 12-1-4.py
请选择模式：
0：字符串链接 1：参数
0
请输入需要查询的学生名字
Tom' UNION SELECT username,username,password FROM user UNION SELECT username,username,password FROM user WHERE username='
[(100, 89, 77), (u'Bill', u'Bill', u'123456'), (u'Lisa', u'Lisa', u'123456'), (u'Tom', u'Tom', u'123456')]
ubuntu@ubuntu:~/sqlite$
```

图 12.16　查询过程与结果

可见，第一次使用参数模式，当我们使用参数构造一个完整的 SQL 查询语句时，新语句依旧不会被执行，而是将我们的输入当作参数进行查询；而第二次使用字符串链接的方式时，我们构造的输入与原 SQL 语句组合成为一条新的语句，新语句除了我们原想查询的三科成绩外，还查询出了 user 表中的账号、密码等信息。这种攻击方式即是大名鼎鼎的 SQL 注入攻击。因此，在使用 sqlite3 模块操作时，切记要使用参数来传递变量！

13. 插入或更新 REPLACE INTO

在操作数据库时，有时无法确定是需要插入新数据还是更新旧数据。此时，我们可以使用 REPLACE INTO 语句。REPLACE INTO 语句会检测设置 UNIQUE 的列，如果新数据

与旧数据在标记了 UNIQUE 列存在重合数据，那么便会更新旧数据；否则将新数据直接插入到数据库中。REPLACE INTO 语句在既可能更新内容，又可能有新内容出现的场合十分常用。

下面我们将使用之前标记过 UNIQUE 的 user 表来学习 REPLACE INTO 语句的应用，如图 12.17 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT * FROM user;
Bill|123456|刘|翔
Lisa|123456|马|德华
Tom|123456|刘|德华
sqlite> REPLACE INTO user(username,password,lastname,firstname)VALUES("Tom",111111,"刘","德华");
sqlite> SELECT * FROM user;
Bill|123456|刘|翔
Lisa|123456|马|德华
Tom|111111|刘|德华
sqlite> REPLACE INTO user(username,password,lastname,firstname)VALUES("Pet",111111,"刘","德华");
sqlite> SELECT * FROM user;
Bill|123456|刘|翔
Lisa|123456|马|德华
Pet|111111|刘|德华
sqlite> REPLACE INTO user(username,password,lastname,firstname)VALUES("Kate",111111,"张","德华");
sqlite> SELECT * FROM user;
Bill|123456|刘|翔
Lisa|123456|马|德华
Pet|111111|刘|德华
Kate|111111|张|德华
sqlite>
```

图 12.17　插入或更新

从图 12.17 中可以看出，第一条 REPLACE INTO 语句中被标记为 UNIQUE 的 username 列存在“Tom”，因此 REPLACE INTO 更新了该数据；第二条 REPLACE INTO 语句中被复合标记为 UNIQUE 的“lastname”“firstname”列的值也存在过，因此第二条也是更新了数据库；而第三条 REPLACE INTO 语句中的值均不与 UNIQUE 的列冲突，因此第三条 REPLACE INTO 将新数据直接插入了数据库。

14. 删除数据 DELETE

删除数据比较简单，只需要使用 DELETE 语句并使用 WHERE 等方式指定删除的行即可：

```
DELETEFROM 表名 WHERE 限定;
```

如果想要删除表中所有数据，则只需去掉 WHERE 限定：

```
DELETE FROM 表名;
```

DELETE 语句的测试如图 12.18 所示。

15. 删除数据表 DROP

DELETE 用于删除表中的数据，如果需要废弃整张表时，只需使用 DROP 语句：

```
DROPTABLE 表名;
```

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT * FROM info;
Tom|15
Kate|17
Bill|19
sqlite> DELETE FROM info WHERE name="Tom";
sqlite> SELECT * FROM info;
Kate|17
Bill|19
sqlite> DELETE FROM info;
sqlite> SELECT * FROM info;
sqlite>
sqlite>
```

图 12.18　删除数据

DROP 语句的测试如图 12.19 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 test.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> .tables
info  rank  user
sqlite> DROP TABLE info;
sqlite> .tables
rank  user
sqlite>
```

图 12.19　删除数据表

12.1.3　SQLite 小结

前面讲述了使用 Python 操作 SQLite 数据库的常用知识，无论是直接操作数据库还是通过 Python 操作数据库，这些基本语句都需要熟练掌握。同时，在使用 Python 操作数据库时，切记当存在变量时要使用参数传递的方式提交；并且操作后使用 commit()方法提交到数据库。

12.2　使用 SQLAlchemy

12.2.1　SQLAlchemy 简介

SQLAlchemy 是 Python 编程语言下的一款开源软件。提供了 SQL 工具包及对象关系映射(ORM)工具。使用 SQLAlchemy，我们不必拘泥于复杂的 SQL 语句，可以像编写程序一样操作数据库。

12.2.2　使用 SQLAlchemy 操作 SQLite 数据库

在开始前，请确保读者的系统中安装了 SQLAlchemy。

在 Windows 32 位下，可以通过安装包安装 setuptools(easy_install)；在 64 位下，可以下载 ez_setup.py 然后在命令行运行 python ez_setup.py 安装并手动将 Python 安装目录

下的 Scripts 的路径添加到系统环境变量 Path 中。然后执行命令 easy_install SQLAlchemy 安装即可。如果安装了 pip 工具，也可以使用 pip install SQLAlchemy 安装。图 12.20 为使用 easy_install 安装。



图 12.20　Windows 下使用 easy_install 安装

在 Ubuntu 下也可以使用上述方法安装，图 12.21 为使用 pip 安装。

```
uibuntu@ubuntu: ~
uibuntu@ubuntu:~$ sudo pip install sqlalchemy
The directory '/home/uibuntu/.cache/pip/http' or its parent directory is not own
ed by the current user and the cache has been disabled. Please check the permiss
ions and owner of that directory. If executing pip with sudo, you may want sudo'
s -H flag.
The directory '/home/uibuntu/.cache/pip' or its parent directory is not owned by
 the current user and caching wheels has been disabled. check the permissions an
d owner of that directory. If executing pip with sudo, you may want sudo's -H fl
ag.
Collecting sqlalchemy
  Downloading SQLAlchemy-1.1.5.tar.gz (5.1MB)
    100% |████████████████████████████████| 5.1MB 13kB/s
Installing collected packages: sqlalchemy
  Running setup.py install for sqlalchemy ... done
Successfully installed sqlalchemy-1.1.5
uibuntu@ubuntu:~$
```

图 12.21　Ubuntu 下使用 pip 安装

安装后，在 Python 命令行中 import sqlalchemy，如果没有报错，则安装成功。

1. 链接数据库

使用 SQLAlchemy 模块时，需要先导入 SQLAlchemy 与 SQLAlchemy.orm 下的所有内容。导入后，定义引擎并指定数据库路径。设置 echo 参数为 True 可以让我们看到 SQLAlchemy 的回显：

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo = True)   # 定义引擎
metadata = MetaData(engine)                                            # 绑定元信息
```

2. 创建数据表

SQLAlchemy 的 orm 工具可以让我们像写一般程序一样操作数据库。我们需要通过

orm 工具建立映射关系。例如 Tabel 类用来建立数据表映射，它的第一个参数为表名，第二个参数为绑定引擎后的元数据，之后的参数为表中得列；而列采用 Column 类进行映射，第一个参数为列名，第二个参数指定数据类型。在建立映射后，使用 Table 对象的 create()方法便可以建立对应数据表：

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo = True)
metadata = MetaData(engine)

user_table = Table(
    'rank',
    metadata,
    Column('id',Integer,primary_key = True),
    Column('name',String(40)),
    Column('classnum',Integer),
    Column('math',Integer),
    Column('cpp',Integer),
    Column('network',Integer),
    )

user_table.create()
```

这段代码的运行效果如图 12.22 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-2.py
2017-02-06 08:24:21,269 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:24:21,269 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:21,269 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:24:21,270 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:21,270 INFO sqlalchemy.engine.base.Engine
CREATE TABLE rank (
        id INTEGER NOT NULL,
        name VARCHAR(40),
        classnum INTEGER,
        math INTEGER,
        cpp INTEGER,
        network INTEGER,
        PRIMARY KEY (id)
)


2017-02-06 08:24:21,271 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:21,275 INFO sqlalchemy.engine.base.Engine COMMIT
ubuntu@ubuntu:~/sqlite$
```

图 12.22　使用 ORM 框架创建数据表

3. 插入数据（不使用 ORM）（不推荐）

在对数据表中的数据执行操作前，我们需要先获取 Table 对象。在上例中我们已经建立了 rank 表，因此不需要重复建表，只需在实例化 rank_table 时将 Table 的 autoload 参数设置为 True 即可。

插入数据时，需要先实例化 insert 对象。得到 insert 对象后，我们只需调用 insert 对象

的 execute()方法并传入想要插入的数据的 json 对象即可(json 对象即用花括号“{}”包围的,以“'键':值”为项,项间使用逗号“,”隔开的格式化数据):

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo = True)
metadata = MetaData(engine)

rank_table = Table('rank',metadata,autoload = True)

insert = rank_table.insert()

insert.execute({'id':1,'name':'Tom','classnum':1,'math':99,'cpp':87,'network':82})
```

这段代码的运行效果如图 12.23 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-3.py
2017-02-06 08:24:56,343 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:24:56,343 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,344 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:24:56,344 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,344 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("rank")
2017-02-06 08:24:56,344 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,345 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:24:56,345 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,346 INFO sqlalchemy.engine.base.Engine PRAGMA foreign_key_list("rank")
2017-02-06 08:24:56,346 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,346 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:24:56,346 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,347 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:24:56,347 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,347 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:24:56,347 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,347 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:24:56,348 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:24:56,349 INFO sqlalchemy.engine.base.Engine INSERT INTO rank (id, name, classnum, math, cpp, network) VALUES (?, ?, ?,
 ?, ?, ?)
2017-02-06 08:24:56,349 INFO sqlalchemy.engine.base.Engine (1, 'Tom', 1, 99, 87, 82)
2017-02-06 08:24:56,349 INFO sqlalchemy.engine.base.Engine COMMIT
ubuntu@ubuntu:~/sqlite$
```

图 12.23　插入数据

验证 rank 表中数据,如图 12.24 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 TestSQLAlchemy.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT * FROM rank;
1|Tom|1|99|87|82
sqlite>
```

图 12.24　验证数据

4. 使用 ORM:创建映射

ORM 的优势在于通过创建类的映射,可以更方便地管理操作。我们可以为表中的数据建立类的映射,并使用 mapper 方法绑定:

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo = True)
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload = True)


class rank(object):

    def __repr__(self):
        return '%s(%r, %r, %r, %r, %r, %r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

r = rank()

print r
```

在这个例子中，我们声明了 rank 类作为 rank 表的一个映射，并重写了__repr__方法用于 print 输出。声明 rank 后，我们使用 mapper 方法将 rank 类与 rank_table 表对象绑定。我们还为这个类创建了实例 r 并将其打印。因为 r 中还没有内容，因此其参数均为 None，如图 12.25 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-4.py
2017-02-06 08:34:41,599 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:34:41,599 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,599 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:34:41,599 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,600 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("rank")
2017-02-06 08:34:41,600 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,601 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:34:41,601 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,601 INFO sqlalchemy.engine.base.Engine PRAGMA foreign_key_list("rank")
2017-02-06 08:34:41,602 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,602 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:34:41,602 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,603 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:34:41,603 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:34:41,603 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:34:41,603 INFO sqlalchemy.engine.base.Engine ()
:34:41,603 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
System Settings ter) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:34:41,603 INFO sqlalchemy.engine.base.Engine ()
rank(None,None,None,None,None,None)
ubuntu@ubuntu:~/sqlite$
```

图 12.25　创建映射

5. 使用 ORM：使用会话操作数据库

在 SQLAlchemy 中，可以使用会话(Session)对象统一操作数据库，更便于开发与维护。

使用 Session 时，首先需要使用 sessionmaker()创建 Session 类并绑定引擎，再实例化 Session 对象操作。操作后，要使用 Session 实例的 flush()方法冲刷缓冲区，所有操作结束

后要使用 commit()方法提交,否则其他 Session 实例得到的仍是上一次 commit()后的数据。

Session 实例的 add 方法用来插入数据,add 方法接收一个表映射类的对象:

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo = True)
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload = True)


class rank(object):

    def __init__(self, uid, name, classnum, math, cpp, network):
        self.id = uid
        self.name = name
        self.classnum = classnum
        self.math = math
        self.cpp = cpp
        self.network = network

    def __repr__(self):
        return '%s(%r, %r, %r, %r, %r, %r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

Session = sessionmaker(bind = engine)
session = Session()

r = rank(2,'Kate',1,86,95,88)

session.add(r)
session.flush()
session.commit()
```

上面这段代码运行后的效果如图 12.26 所示。

验证数据库中插入了新数据,如图 12.27 所示。

Session 实例的 query()方法用来返回一个查询实例,它接收一个表映射类作为参数。查询对象的 first()用来返回第一条结果,all()用来返回一个结果列表,而 one()当恰好一条结果时返回,否则会抛出异常:

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-5.py
2017-02-06 08:52:27,577 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:52:27,577 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,578 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
2017-02-06 08:52:27,578 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,578 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("rank")
2017-02-06 08:52:27,579 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,580 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:52:27,580 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,580 INFO sqlalchemy.engine.base.Engine PRAGMA foreign_key_list("rank")
2017-02-06 08:52:27,580 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,581 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:52:27,581 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,581 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:52:27,582 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,582 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-06 08:52:27,582 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,582 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-06 08:52:27,582 INFO sqlalchemy.engine.base.Engine ()
2017-02-06 08:52:27,585 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2017-02-06 08:52:27,586 INFO sqlalchemy.engine.base.Engine INSERT INTO rank (id, name, classnum, math, cpp, network) VALUES (?, ?, ?,
 ?, ?, ?)
2017-02-06 08:52:27,586 INFO sqlalchemy.engine.base.Engine (2, 'Kate', 1, 86, 95, 88)
2017-02-06 08:52:27,587 INFO sqlalchemy.engine.base.Engine COMMIT
ubuntu@ubuntu:~/sqlite$
```

图 12.26　使用会话操作数据库

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ sqlite3 TestSQLAlchemy.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT * FROM rank;
1|Tom|1|99|87|82
2|Kate|1|86|95|88
sqlite>
```

图 12.27　验证数据

```
#!/usr/bin/python
# coding=utf-8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db', echo=True)
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload=True)


class rank(object):

    def __init__(self, uid, name, classnum, math, cpp, network):
            self.id = uid
            self.name = name
            self.classnum = classnum
            self.math = math
            self.cpp = cpp
            self.network = network

    def __repr__(self):
```

```
        return '%s(%r,%r,%r,%r,%r,%r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

Session = sessionmaker(bind=engine)
session = Session()

r = session.query(rank)

print "*****************"
print r
print r.all()
```

这段代码的运行效果如图 12.28 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-6.py
2017-02-07 01:30:22,374 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
2017-02-07 01:30:22,374 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,375 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
2017-02-07 01:30:22,375 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,376 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("rank")
2017-02-07 01:30:22,376 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,377 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-07 01:30:22,377 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,378 INFO sqlalchemy.engine.base.Engine PRAGMA foreign_key_list("rank")
2017-02-07 01:30:22,378 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,378 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-07 01:30:22,378 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,379 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-07 01:30:22,379 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,380 INFO sqlalchemy.engine.base.Engine PRAGMA index_list("rank")
2017-02-07 01:30:22,380 INFO sqlalchemy.engine.base.Engine ()
2017-02-07 01:30:22,380 INFO sqlalchemy.engine.base.Engine SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sq
lite_temp_master) WHERE name = 'rank' AND type = 'table'
2017-02-07 01:30:22,380 INFO sqlalchemy.engine.base.Engine ()
*****************
SELECT rank.id AS rank_id, rank.name AS rank_name, rank.classnum AS rank_classnum, rank.math AS rank_math, rank.cpp AS rank_cpp, rank
.network AS rank_network
FROM rank
2017-02-07 01:30:22,384 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2017-02-07 01:30:22,385 INFO sqlalchemy.engine.base.Engine SELECT rank.id AS rank_id, rank.name AS rank_name, rank.classnum AS rank_c
lassnum, rank.math AS rank_math, rank.cpp AS rank_cpp, rank.network AS rank_network
FROM rank
2017-02-07 01:30:22,385 INFO sqlalchemy.engine.base.Engine ()
[rank(1,u'Tom',1,99,87,82), rank(2,u'Kate',1,86,95,88)]
ubuntu@ubuntu:~/sqlite$
```

图 12.28　使用 ORM 查询

在本例中，我们先输出了查询实例，然后将其输出，因为其重载了__repr__方法，所以我们输出了一条 SQL SELECT 语句。而 all()返回的才是查询的结果。

查询实例同样提供了丰富的函数用于特殊条件查询。limit 函数用于指定查询条目；offset 函数用于指定查询起始偏移；order_by 函数用于排序；filter 用于筛选，相当于 WHERE。开启 echo 回显可以清晰地观察每条代码所对应的 SQL 语句(下例为了清晰，关闭了回显)：

```
#!/usr/bin/python
# coding=utf-8

from sqlalchemy import *
from sqlalchemy.orm import *
```

```
engine = create_engine('sqlite:///./TestSQLAlchemy.db')
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload=True)


class rank(object):

    def __init__(self, uid, name, classnum, math, cpp, network):
            self.id = uid
            self.name = name
            self.classnum = classnum
            self.math = math
            self.cpp = cpp
            self.network = network

    def __repr__(self):
        return '%s(%r, %r, %r, %r, %r, %r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

Session = sessionmaker(bind=engine)
session = Session()

print session.query(rank).all()
print session.query(rank).limit(3).all()
print session.query(rank).offset(2).all()
print session.query(rank).order_by('cpp').all()
print session.query(rank).filter(rank.cpp>90).all()
```

这段代码(为了清晰关闭了 echo 回显)的运行效果如图 12.29 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python  12-2-7.py
[rank(1,u'Tom',1,99,87,82), rank(2,u'Kate',1,86,95,88), rank(3,u'Bill',2,77,89,67), rank(4,u'Mark',2,91,92,88)]
[rank(1,u'Tom',1,99,87,82), rank(2,u'Kate',1,86,95,88), rank(3,u'Bill',2,77,89,67)]
[rank(3,u'Bill',2,77,89,67), rank(4,u'Mark',2,91,92,88)]
[rank(1,u'Tom',1,99,87,82), rank(3,u'Bill',2,77,89,67), rank(4,u'Mark',2,91,92,88), rank(2,u'Kate',1,86,95,88)]
[rank(2,u'Kate',1,86,95,88), rank(4,u'Mark',2,91,92,88)]
ubuntu@ubuntu:~/sqlite$
```

图 12.29　使用 ORM 中的子句

可见,查询实例的各种方法与我们直接写 SQL 语句的查询效果相同。相比之下,ORM 方式更加方便简洁、易于维护。

得到表映射类的对象实例后,我们可以使用 update 方法更新其值。结合使用 filter() 与 update()方法,便可以像使用 SQL 中的 UPDATE、WHERE 一样来更新值。例如,我们将“Bill”的数学改为 100 分:

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db')
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload = True)


class rank(object):

    def __init__(self, uid, name, classnum, math, cpp, network):
            self.id = uid
            self.name = name
            self.classnum = classnum
            self.math = math
            self.cpp = cpp
            self.network = network

    def __repr__(self):
        return '%s(%r,%r,%r,%r,%r,%r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

Session = sessionmaker(bind = engine)
session = Session()

session.query(rank).filter(rank.name == "Bill").update({rank.math:100})
print session.query(rank).filter(rank.name == "Bill").one()

session.flush()
session.commit()
```

这段代码的运行效果如图 12.30 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-8.py
rank(3,u'Bill',2,100,89,67)
ubuntu@ubuntu:~/sqlite$
```

图 12.30 使用 ORM 更新数据

同样,可以使用表映射类的实例的 delete 方法删除一条数据:

```
#!/usr/bin/python
# coding = utf - 8

from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('sqlite:///./TestSQLAlchemy.db')
metadata = MetaData(engine)

rank_table = Table('rank', metadata, autoload = True)


class rank(object):

    def __init__(self, uid, name, classnum, math, cpp, network):
                self.id = uid
                self.name = name
                self.classnum = classnum
                self.math = math
                self.cpp = cpp
                self.network = network

    def __repr__(self):
        return '%s(%r, %r, %r, %r, %r, %r)' % (self.__class__.__name__, self.id,
self.name, self.classnum, self.math, self.cpp, self.network)

mapper(rank, rank_table)

Session = sessionmaker(bind = engine)
session = Session()

print session.query(rank).all()
session.query(rank).filter(rank.name == "Bill").delete()
print session.query(rank).all()

session.flush()
session.commit()
```

这段代码的运行效果如图 12.31 所示。

```
ubuntu@ubuntu: ~/sqlite
ubuntu@ubuntu:~/sqlite$ python 12-2-9.py
[rank(1,u'Tom',1,99,87,82), rank(2,u'Kate',1,86,95,88), rank(3,u'Bill',2,100,89,67), rank(4,u'Mark',2,91,92,88)]
[rank(1,u'Tom',1,99,87,82), rank(2,u'Kate',1,86,95,88), rank(4,u'Mark',2,91,92,88)]
ubuntu@ubuntu:~/sqlite$
```

图 12.31　使用 ORM 删除数据

12.2.3 SQLAlchemy 小结

通过 SQLAlchemy,我们可以像写一般 Python 程序一样操作数据库而不必拘泥于复杂的 SQL 语句。同时,使用 ORM 工具,可以更加清晰、规范地管理数据库,也大大降低了未来维护的复杂度。

ORM 工具在数据库操作时十分常用,因此读者应该至少掌握一种支持 ORM 的工具,以应对未来的需求。

习 题 12

一、选择题

1. Python 中 sqlite3 模块使用(　　)函数连接数据库。

A. connect　　B. commit　　C. session　　D. sqlite

2. SQLAlchemy 将数据库的一个表映射为 Python 中的一个(　　)。

A. 函数　　B. 类　　C. 对象　　D. 过程

3. 当 SQLAlchemy 的一个会话结束后,我们需要使用(　　)方法提交会话,才能保证数据被正确地写入数据库。

A. connect　　B. commit　　C. session　　D. sqlite

二、填空题

1. 在 SQL 中,创建数据表使用________命令;查询数据库使用________命令;插入数据使用________命令;更新数据使用________命令;删除数据使用________命令;删除数据表使用________命令。

2. SQL 中________可以用来连接子句;________子句用来对数据排序,它还可以通过________、________子句进一步限制每次查询的条数以及起始位置。

3. SQL 中的主键是________。

三、论述题

1. 简述 SQL 中的约束语句以及它们的作用。

2. 简述 ORM 工具的优缺点并使用 ORM 工具制作简单数据库管理工具。

四、编程题

1. 改写第 6 章成绩排序的程序,使用数据库存储学生姓名与成绩。

2. 编写命令行下的笔记程序,它需要有如下功能:a. 将输入存储到数据库;b. 从数据库中查询所有笔记;c. 显示对应的笔记;d. 查看最近五条笔记。使用数据库保证程序重新启动后仍可以从数据库中获取之前存储的笔记信息。

第 13 章　Python Socket 网络编程

13.1　Socket 简介

13.1.1　Socket 通信概述

Socket(又名套接字)是进程通信的一种方式。Socket 不仅仅可以在本地进程间通信,还可以依照 TCP/IP 协议在网络主机的进程间通信,即通过 IP 地址与端口号建立 Socket 连接进行通信。在 TCP/IP 网络应用中,通信的两个进程间相互作用的主要模式是客户/服务器(Client/Server,C/S)模式,即客户向服务器发出服务请求,服务器接收到请求后,提供相应的服务。



视频讲解

13.1.2　TCP 协议与 UDP 协议的区别

TCP 与 UDP 是两种常用的传输层协议,也是 Socket 通信中常用的两种传输层协议,了解二者的区别对编写 Socket 网络应用程序很重要。

TCP 协议是传输控制协议,提供面向连接、可靠的字节流服务。使用 TCP 协议通信时,客户端与服务端通过三次"握手"建立连接,四次"握手"关闭连接等方式保证通信的可靠性。TCP 提供超时重发,丢弃重复数据,检验数据,流量控制等功能,保证数据能从一端传到另一端。因为 TCP 通过各种方式保证了可靠性,所以其速度较 UDP 通信慢一些,报文较 UDP 通信大一些。

UDP 协议是用户数据报协议,它是一个简单的面向数据报的运输层协议。UDP 不提供可靠性,它只是把应用程序传给 IP 层的数据报发送出去,但是并不能保证它们能到达目的地。由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接,且没有超时重发等机制,故而传输速度很快。虽然 UDP 在速度和报文大小上均具有优势,但是 UDP 不保证报文是否丢失或者多个报文的顺序正确与否等可靠性问题。

13.2　Python Socket 编程

13.1 节中,我们简单介绍了 Socket 通信与 TCP、UDP 协议,本节我们将介绍如何使用 Python 编写 Socket 程序。

Python 自带了 Socket 模块,提供了强大且便捷的 Socket 开发接口。

13.2.1 简易 Socket 通信

Socket 通信采用 C/S 模式,因此我们需要分别编写客户端与服务端程序进行通信。

使用 Socket 模块进行通信时,我们首先需要实例化一个套接字实例并指定套接字类型与协议类型等,Socket 模块支持的套接字类型如表 13.1 所示。

表 13.1 套接字类型

Socket 类型	描　　述
socket.AF_UNIX	只能够用于单一的 UNIX 系统进程间通信
socket.AF_INET	服务器之间的网络通信
socket.AF_INET6	服务器之间的 IPv6 网络通信
socket.SOCK_STREAM	流式 Socket(TCP 协议格式)
socket.SOCK_DGRAM	数据报式 Socket(UDP 协议格式)
socket.SOCK_RAW	原始套接字,用于处理 ICMP、IGMP 等网络报文等特殊报文或者定义 IP 头的报文
socket.SOCK_SEQPACKET	可靠的连续数据包服务

因此,创建 TCP Socket 时,我们要使用 socket(socket.AF_INET, socket.SOCK_STREAM),而创建 UDP Socket 时使用 socket(socket.AF_INET, socket.SOCK_DGRAM)。

实例化 Socket 对象后,我们需要调用其实例方法来进行 Socket 通信,Socket 实例有以下常用的方法,如表 13.2 所示。

表 13.2 Socket 常用方法

Socket 方法	描　　述
服务端 Socket 方法	
bind(address)	将套接字绑定到地址,在 AF_INET 下,以元组(host,port)的形式表示地址
listen(backlog)	开始监听 TCP 传入连接。backlog 指定在拒绝连接之前,操作系统可以挂起的最大连接数量,该值至少为 1
accept()	接受 TCP 连接并返回(conn,address),其中 conn 是新的套接字对象,可以用来接收和发送数据。address 是连接客户端的地址
客户端 Socket 方法	
connect(address)	连接到 address 处的套接字。一般 address 的格式为元组(hostname,port),如果连接出错,返回 socket.error 错误
connect_ex(adddress)	功能与 connect(address)相同,但是成功返回 0,失败返回 errno 的值
公共 Socket 方法	
recv(bufsize[,flag])	接收 TCP 套接字的数据。数据以字符串形式返回,bufsize 指定要接收的最大数据量。flag 提供有关消息的其他信息,通常可以忽略
end(string[,flag])	发送 TCP 数据。将 string 中的数据发送到连接的套接字。返回值是要发送的字节数量,该数量可能小于 string 的字节大小
sendall(string[,flag])	完整发送 TCP 数据。将 string 中的数据发送到连接的套接字,但在返回之前会尝试发送所有数据。成功返回 None,失败则抛出异常

续表

Socket 方法	描　　述
公共 Socket 方法	
recvfrom(bufsize[. flag])	接收 UDP 套接字的数据。与 recv()类似,但返回值是(data,address)。其中 data 是包含接收数据的字符串,address 是发送数据的套接字地址
sendto (string [, flag], address)	发送 UDP 数据。将数据发送到套接字,address 是形式为(ipaddr,port)的元组,指定远程地址。返回值是发送的字节数
close()	关闭套接字
getpeername()	返回连接套接字的远程地址。返回值通常是元组(ipaddr,port)
getsockname()	返回套接字自己的地址。通常是一个元组(ipaddr,port)
setsockopt(level,optname,value)	设置给定套接字选项的值
getsockopt (level, optname [. buflen])	返回套接字选项的值
settimeout(timeout)	设置套接字操作的超时期,timeout 是一个浮点数,单位是秒。值为 None 表示没有超时期。一般,超时期应该在刚创建套接字时设置,因为它们可能用于连接的操作(如 connect())
gettimeout()	返回当前超时期的值,单位是秒,如果没有设置超时期,则返回 None
fileno()	返回套接字的文件描述符
setblocking(flag)	如果 flag 为 0,则将套接字设为非阻塞模式,否则将套接字设为阻塞模式(默认值)。非阻塞模式下,如果调用 recv()没有发现任何数据,或 send()调用无法立即发送数据,那么将引起 socket. error 异常
makefile()	创建一个与该套接字相关联的文件

从表 13.2 中我们可以看出 TCP 协议下的 Socket 与 UDP 协议下 Socket 的不同之处。由于 TCP 需要建立连接后再发送数据,因此 TCP 的 Socket 建立连接后不需要再指定发送地址;而 UDP 的 Socket 因为不需要握手即可发送数据,因此不需要绑定连接,而是每次发送都要指定发送地址。

网络通信是相当复杂的,因为各式各样的环境变化会引起不同的异常,Socket 模块提供了 4 种异常,如表 13.3 所示。

表 13.3　Socket 异常类

异 常 类 型	描　　述
socket. error	由 Socket 相关错误引发
socket. herror	由地址相关错误引发
socket. gaierror	由地址相关错误,如 getaddrinfo()或 getnameinfo()引发
socket. timeout	当 Socket 出现超时时引发。超时时间由 settimeout()提前设定

接下来我们将分别讲解使用 TCP 或 UDP 协议的 Socket 程序。

1. TCP Socket

因为 Socket 程序采用 C/S 的模式,我们将分别讲解服务端与客户端的一般流程。

服务端使用 Socket 模块编写 TCP Socket 程序一般需要如下步骤:

(1) 导入 Socket 模块；
(2) 创建 Socket 实例并指定类型；
(3) 将 Socket 实例绑定到地址；
(4) 开始监听 TCP 连接，等待客户端连接；
(5) 接受 TCP 连接；
(6) 开始收发数据；
(7) 关闭 TCP 连接；
(8) 关闭 Socket 实例。

而客户端逻辑则较为简单：

(1) 导入 Socket 模块；
(2) 创建 Socket 实例并指定类型；
(3) 向服务端地址请求连接；
(4) 开始收发数据；
(5) 关闭 Socket 实例。

因为服务端可能与多个客户端通信，因此服务端一般对每个连接(连接也是 Socket 实例)进行操作；而客户端只需要与一个服务端通信，因此只需要操作当前套接字即可。

下面我们就来演示如何编写简单的 Socket 通信程序(两个程序均在本地运行，本地 IP 地址为 127.0.0.1)：

服务端：

```
from socket import *

sock = socket(AF_INET, SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 12345
BUFFER_SIZE = 4096
ADDR = (HOST, PORT)

sock.bind(ADDR)

sock.listen(5)

_close = False

while True:
    if(_close):
        break
    print 'Waiting for connection...'
    conn, addr = sock.accept()
    print 'Get connection from :', addr
```

```
    while True:

        data = conn.recv(BUFFER_SIZE)

        if not data:
            continue
        if(data == 'Bye'):
            conn.close()
            break
        elif(data == 'Close'):
            _close = True
            conn.close()
            sock.close()
            break
        else:
            mess = 'Get message from [ %s] : %s' % (addr, data)
            print mess
            conn.send('Got it !')
```

客户端代码中,我们实现了之前介绍的服务端完整的步骤。为了能够根据指令关闭连接与套接字对象,我们对收到的数据进行了额外的判断,如果是'Bye'则会关闭与当前客户端的连接,之后可以使用其他客户端重新连接;而如果是'Close'则会关闭主套接字,不再接受任何连接。

客户端:

```
from socket import *

sock = socket(AF_INET, SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 12345
BUFFER_SIZE = 4096
ADDR = (HOST, PORT)

sock.connect(ADDR)

while True:
    data = raw_input("Please input your message:\n")
    if not data:
        continue
    sock.send(data)
    if(data == 'Bye' or data == 'Close'):
        sock.close()
        break
    data = sock.recv(BUFFER_SIZE)
    if not data:
        continue
    print 'Get message from server: %s' % data
```

当运行样例时，我们需要先打开服务端程序，再使用客户端程序连接。例如，我们先后使用两个客户端程序连接服务端，第一个客户端发送数据后通知服务端关闭当前连接，第二个客户端发送数据后通知关闭主套接字不再接受连接：

先后两个客户端输出如图13.1所示。

```
C:\WINDOWS\system32\cmd.exe

F:\Python\第十三章>python 13-2-2.py
Please input your message:
I love Python !
Get message from server: Got it !
Please input your message:
Bye

F:\Python\第十三章>python 13-2-2.py
Please input your message:
I love Python, too !
Get message from server: Got it !
Please input your message:
Close

F:\Python\第十三章>
```

图13.1　客户端输出

服务端输出如图13.2所示。

```
C:\WINDOWS\system32\cmd.exe

F:\Python\第十三章>python 13-2-1.py
Waiting for connection...
Get connection from : ('127.0.0.1', 11913)
('Get message from [127.0.0.1] : 11913', 'I love Python !')
Waiting for connection...
Get connection from : ('127.0.0.1', 11921)
('Get message from [127.0.0.1] : 11921', 'I love Python, too !')

F:\Python\第十三章>
```

图13.2　服务端输出

我们已经实现了TCP Socket互发消息的简单程序。

2. UDP Socket

相比TCP Socket，UDP Socket就要简单得多，因为UDP Socket不需要建立连接后通信，每次发送时需要指定地址，所以代码简单得多。

服务端使用Socket模块编写UDP Socket程序一般需要如下步骤：

(1) 导入Socket模块；

(2) 创建Socket实例并绑定地址；

(3) 等待收发信息；

(4) 关闭Socket实例。

客户端逻辑：

(1) 导入Socket模块；

(2) 创建Socket实例并绑定地址；

(3) 收发信息；

(4) 关闭 Socket 实例。

下面我们编写一个简单的 UDP Socket 通信程序：

服务端：

```
from socket import *

HOST = '127.0.0.1'
PORT = 12345
ADDR = (HOST,PORT)
BUFFER_SIZE = 4096

sock = socket(AF_INET,SOCK_DGRAM)

sock.bind(ADDR)

while True:
    print 'Waiting for messages...'
    data,addr = sock.recvfrom(BUFFER_SIZE)
    if(data == 'Close'):
        sock.close()
        break
    else:
        print 'Get message from',addr,' : ',data
```

客户端：

```
from socket import *

HOST = '127.0.0.1'
PORT = 12345
ADDR = (HOST, PORT)

sock = socket(AF_INET, SOCK_DGRAM)

while True:
    data = raw_input('Please input your message : \n')
    sock.sendto(data, ADDR)
    if(data == 'Close'):
        sock.close()
        break
```

同样，我们首先需要打开服务端，再使用客户端进行连接：

客户端输出如图 13.3 所示。

服务端输出如图 13.4 所示。

13.2.2 使用多线程的多端 Socket 通信

在之前的例子中，我们实现了简单的 Socket 通信，但是，这种方式使服务端同时只能与

```
F:\Python\第十三章>python 13-2-4.py
Please input your message :
I love Python !
Please input your message :
Close

F:\Python\第十三章>
```

图 13.3　客户端输出

```
F:\Python\第十三章>python 13-2-3.py
Waiting for messages...
Get message from ('127.0.0.1', 50759)  :  I love Python !
Waiting for messages...

F:\Python\第十三章>
```

图 13.4　服务端输出

一个客户端通信，而且如果客户端始终不向服务端发送数据时，服务端的线程会被阻塞。在实际场景中，一个服务端往往需要同时处理多个客户端的请求。

既然单一线程工作时服务端线程会因 I/O 被阻塞，那么最简单的解决方案就是多线程 Socket 通信。

以 TCP Socket 为例，服务端每 accept()到一个新的连接时，就会创建一个子线程用来处理这个连接 I/O，即使每个子线程中的 I/O 仍然是阻塞的，但是总体上看我们已经能够实现单服务端多客户端的 Socket 通信。

根据这个解决方案，我们可以编写服务端如下代码：

```
from socket import *
from threading import *


def dealSocket(conn, addr, sid):
    global BUFFER_SIZE
    while True:

        data = conn.recv(BUFFER_SIZE)

        if not data:
            continue
        if(data == 'Bye'):
            conn.close()
            break
        else:
            mess = 'Get message from connection_%d[%s] : %s' % (sid,addr,data)
            print mess
```

```
            conn.send('Got it !')
    pass

sock = socket(AF_INET, SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 12345
BUFFER_SIZE = 4096
ADDR = (HOST, PORT)

sock.bind(ADDR)

sock.listen(5)

conn_num = 0

while True:

    print 'Waiting for connection...'
    conn, addr = sock.accept()
    print 'Get connection from :', addr
    conn_num += 1
    Thread(target=dealSocket, args=(conn, addr, conn_num)).start()
```

运行服务端代码，因为客户端仍与单服务端通信，因此我们可以使用之前的 TCP Socket 客户端代码进行测试。我们使用三个客户端同时连接到服务端并与其通信，得到服务端输出如图 13.5 所示（客户端操作省略，只需要看服务端）。

```
C:\Python27\python.exe
Waiting for connection...
Get connection from : ('127.0.0.1', 3134)
Waiting for connection...
Get message from connection_1 [('127.0.0.1', 3134)] : Hi! I'm Client_1!
Get connection from : ('127.0.0.1', 3137)
Waiting for connection...
Get message from connection_2 [('127.0.0.1', 3137)] : Hi! I'm Client_2!
Get message from connection_1 [('127.0.0.1', 3134)] : I'm Client_1! Do you miss me?
Get message from connection_2 [('127.0.0.1', 3137)] : I'm Client_2! Do you miss me?
```

图 13.5　多线程 Socket 服务端

所以我们可以使单服务端同时与多客户端通信。

13.2.3　基于 select、poll 或 epoll 的异步 Socket 通信

当客户端很少时，多线程的方式可以很好地解决多端通信问题。而当有几百个甚至几千个客户端时，无论是多线程还是多进程，都会在线程或进程切换时浪费大量资源。这时，我们需要使用 select、poll 或 epoll 等异步 Socket 方式。

select 最早于 1983 年出现在 4.2BSD 中，它通过一个 select()系统调用来监视多个文件描述符的数组，当 select()返回后，该数组中就绪的文件描述符便会被内核修改标识位，使

得进程可以获得这些文件描述符从而进行后续的读写操作。select 目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。

使用 select 时，需要导入 select 模块。服务端 Socket 对象开始监听后，使用 select 模块下的 select 方法，该方法接收 4 个参数，并返回三个列表。第一个参数表示 select 监听的连接列表，该列表中的连接如果活动则会加入到第一个返回列表中；第二个参数的列表会被完整地传回到第二个返回列表；第三个参数中发生错误的连接会被加入第三个返回列表中；第四个参数为 select 监听的频率，单位是：次每秒。

通过操作这三个返回列表，我们可以方便地管理 Socket 连接。

我们使用 select 实现一个简单的多客户端异步 Socket 服务端：

```
# -*- coding=utf-8 -*-
from socket import *
import select

HOST = '127.0.0.1'
PORT = 12345
BUFFER_SIZE = 4096
ADDR = (HOST, PORT)

sock = socket(AF_INET, SOCK_STREAM)
sock.bind(ADDR)

sock.listen(5)

#设置 Socket 为非阻塞模式
sock.setblocking(False)

inputs = [sock, ]
outputs = []

message_dict = {}

while True:

    r_list, w_list, e_list = select.select(inputs, outputs, inputs, 1)
    # 使用 select 监听 Socket 连接
    # 第一个参数是 select 监听的连接,其中活动的连接会传给 r_list
    # 第二个参数会被完整地传给 w_list
    # 第三个参数中错误的连接会被传送给 e_list
    # 第四个参数表示监听的频率,单位为秒

    print('Amount of sockets : %d' % len(inputs))
    print(r_list)

    for conn in r_list:
        if conn == sock:
```

```
        # 当 sock 活动时,说明有新连接接入
                conn, address = conn.accept()
                inputs.append(conn)
                message_dict[conn] = []
            else:
                # 其他连接活动时,说明原有连接有新消息
                try:
                    data = conn.recv(BUFFER_SIZE)
                except Exception as ex:
                    # 异常处理,若客户端关闭连接
                    inputs.remove(conn)
                else:
                    message_dict[conn].append(data)
                    outputs.append(conn)

        # w_list 中保存给我发过消息的连接
        for conn in w_list:
            data = message_dict[conn][0]
            print data
            del message_dict[conn][0]
            conn.send("Got it !")
            outputs.remove(conn)

        for conn in e_list:
        # 如果连接发生异常,不再监听该连接
            inputs.remove(conn)
```

本例的关键步骤解释可见注释,总体思路是分别遍历三个 select 的返回列表,对第一个活动列表做判断,如果是服务端 sock 对象,则是有新连接接入,否则是原有连接有新消息;对于第二个列表我们仅用其存储发送过消息的连接,并遍历这些连接对应的消息字典输出消息;对于第三个返回列表,其全为发生异常的连接,我们只需要不再监听这些连接即可。

我们可以使用之前的 TCP Socket 客户端代码来测试本例并观察输出。

select 的方式同样存在缺点。select 的缺点之一是单个进程能够监视的文件描述符的数量存在最大限制,在 Linux 上一般为 1024,不过可以通过修改宏定义甚至重新编译内核的方式提升这一限制。另外,select 所维护的存储大量文件描述符的数据结构,随着文件描述符数量的增大,其复制的开销也线性增长。同时,由于网络响应时间的延迟使得大量 TCP 连接处于非活跃状态,但调用 select 会对所有 Socket 进行一次线性扫描,所以这也浪费了一定的开销。poll 方式本质上与 select 没有区别,它将用户传入的数组拷贝到内核空间,然后查询每个文件描述符对应的设备状态,如果设备就绪则在设备等待队列中加入一项并继续遍历,如果遍历完所有文件描述符后没有发现就绪设备,则挂起当前进程,直到设备就绪或者主动超时,被唤醒后它又要再次遍历文件描述符。这个过程经历了多次无谓的遍历。它没有最大连接数的限制,原因是它是基于链表来存储的,但是同样有一个缺点:大量的文件描述符的数组被整体复制于用户态和内核地址空间之间,而不管这样的复制是不是有意义。poll 还有一个特点是“水平触发”,如果报告了文件描述符后,没有被处理,那么下

次 poll 时会再次报告该文件描述符。

无论是 select 还是 poll，其内部都会遍历所有连接的文件描述符状态，因此当连接数很多时，这种遍历会使效率线性地下降。为了解决这个问题，我们可以使用 epoll 的方式。

epoll 除了提供 select/poll 提供的 IO 事件的水平触发(Level Triggered)外，还提供了边缘触发(Edge Triggered)。很多情况下，虽然连接数很多，但是同一时间可能只有部分的 Socket 是“活跃”的，但是 select/poll 每次调用都会线性扫描全部的集合，导致效率呈现线性下降。而 epoll 不存在这个问题，它只会对“活跃”的 Socket 进行操作。这是因为在内核实现中 epoll 是根据每个文件描述符上面的 callback 函数来实现的。那么，只有“活跃”的 Socket 才会主动地去调用 callback 函数，其他 Socket 则不会。除此之外，epoll 相较于 select 还有其他好处：epoll 理论上没有最大连接数的限制，其实际最大连接数即为系统可同时打开的最多文件的数目，这个数字一般远大于 select 的限制，在 1GB 内存的机器上大约是 10 万左右，一般来说这个数目和系统内存关系很大。

epoll 相对于 select 的方式的缺点为非 Linux 平台对其支持性不是很好，例如 Windows 目前仍不支持 epoll 而采用完成端口的方式。

在 Python 中使用 epoll 编程时，需要实例化一个 epoll 对象。对于每个连接，需要使用 epoll 实例的 register()方法注册关注，该方法需要两个参数，分别为连接的文件描述符整数代号 fileno 与触发时刻，触发时刻有 EPOLLIN 与 EPOLLOUT 等，分别表示输入、输出活动。查询活动的连接时，我们可以使用 epoll 实例的 poll()方法，该方法需要一个参数，为查询几秒内的活动，该方法返回一个列表，该列表是一个元组的集合，每个元组有两个元素，分别为文件描述符代号 fileno 与活动事件整型代号。当一个被 epoll 监听的连接需要改变监听的活动时，可以使用 epoll 实例的 modify 方法，该方法接受两个参数分别为 fileno 与活动类型，如果活动类型为 0 则不再监听。此外，epoll 还会自动关注 EPOLLHUP 活动，该活动表示连接被挂起。当一个连接不再需要监听时，可以使用 epoll 实例的 unregister 方法注销该连接的文件描述符。

我们使用 epoll 的方式编写一个简单的多客户端异步 Socket 服务端代码，并在 Linux 系统下运行测试(由于 Windows 不支持 epoll，故 Python 在运行时会报错；Windows 10 下可以开启内嵌 Linux 子系统，在 bash 下运行)：

```
# -*- coding=utf-8 -*-

import select
import socket


HOST = '127.0.0.1'
PORT = 12345
BUFFER_SIZE = 4096
ADDR = (HOST, PORT)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(ADDR)
```

```
sock.listen(10)
sock.setblocking(0)

epoll = select.epoll()

epoll.register(sock.fileno(), select.EPOLLIN)

# 字典 connections 映射文件描述符(整数)到其相应的网络连接对象
connections = {}
requests = {}
responses = {}

while True:

    events = epoll.poll(1)

    for fileno, event in events:
        # 如果是服务端产生 event,表示有一个新的连接进来
        if fileno == sock.fileno():
            connection, address = sock.accept()
            print('client connected:', address)

            connection.setblocking(0)

            # 为新的 socket 注册 epoll 关注
            epoll.register(connection.fileno(), select.EPOLLIN)
            connections[connection.fileno()] = connection
            # 初始化接收的数据
            requests[connection.fileno()] = ''

        # 如果发生一个输入 event
        elif event == select.EPOLLIN:
            # 接收客户端发送过来的数据
            requests[fileno] += connections[fileno].recv(BUFFER_SIZE)
            # 如果客户端退出,关闭客户端连接,取消所有的读和写监听
            if not requests[fileno]:
                connections[fileno].close()
                del connections[fileno]
                del requests[connections[fileno]]
                print(connections, requests)
                epoll.modify(fileno, 0)
            else:
                # 接收数据后,将监听模式修改为 EPOLLOUT 监听输出活动
                epoll.modify(fileno, select.EPOLLOUT)
                print(requests[fileno])

        # 如果发生一个输出 event
        elif event == select.EPOLLOUT:
            connections[fileno].send('Got it !')
```

```
                # 发送数据后,将监听模式修改为 EPOLLIN 监听输入活动
                epoll.modify(fileno, select.EPOLLIN)

            # 如果发生一个 EPOLLHUP event,则关闭连接
            elif event == select.EPOLLHUP:
                print("A connection closed !")
                epoll.unregister(fileno)
                connections[fileno].close()
                del connections[fileno]
```

我们在 Ubuntu 系统上测试,服务端与两客户端如图 13.6 所示。

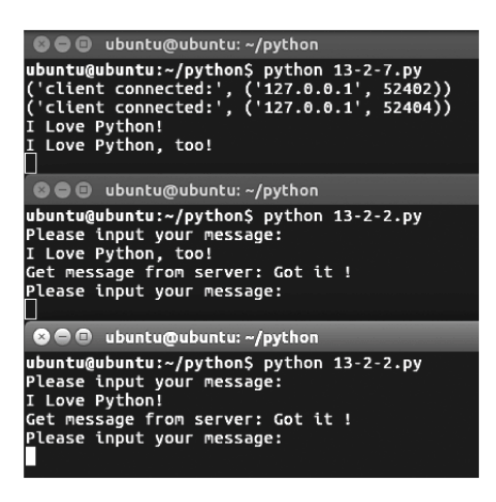


图 13.6　epoll 服务端测试

习　题　13

一、选择题

1. (　　)协议可以保证报文发送的正确性。

   A. UDP　　B. IP　　C. TCP　　D. tftp

2. 使用 Socket 通信前,我们需要指定通信协议、IP 地址及(　　)。

   A. 端口号　　B. 目标计算机名

   C. 目标计算机 MAC 地址　　D. DNS 服务器地址

3. 在等待消息时服务端或客户端被阻塞,这种实现被称为(　　)Socket 通信。

   A. 异步　　B. 同步　　C. 多线程　　D. 多进程

二、填空题

1. Python 中 Socket 模块服务端的________方法用于绑定地址,________方法用于监听,________方法用于接受客户端连接。

2. Python 中 Socket 模块客户端的________方法用于连接服务端。

3. Python 中 Socket 客户端与服务端都可以使用________发送消息,使用________接收消息。

三、论述题

1. 简述 TCP 协议与 UDP 通信协议的异同。

2. 分别简述 Python 中 Socket 模块使用 TCP 与 UDP 协议时，服务端与客户端的流程。

四、编程题

分别使用多线程、select、poll 与 epoll 编写简单的 Socket 服务端，实现多人聊天室程序（一人发送消息到服务端，服务端向所有连接的客户端转发消息）。

第14章 Python Web 编程

14.1 Python Web 编程简介

Python 语言既具有脚本语言开发的快捷性，也因其对 OOP 的支持具有维护的便捷性。除此之外，Python 丰富的拓展使其还能胜任大数据挖掘、科学计算等方向的任务。在近几年数据科学盛行的环境下，越来越受到开发者的青睐。



视频讲解

同时，目前基于 Python 开发的 Web 框架功能越来越丰富。例如 Django，它是一个被广泛应用的十分全能的大型 Web 框架。熟悉 Django 的开发者可以十分快速地开发出网站原型。我们接下来要介绍的 Flask 框架同样是 Python 下十分流行的 Web 框架，它与 Django 框架相比具有更加轻量级、简洁、易拓展的优势，更加适合初学者上手。

近几年，使用 Python 编写 Web 业务应用的企业越来越多。同时，Web 应用也是技术的"大杂烩"。通过简单了解 Web 应用的开发，读者可以在实际应用中复习之前学过的很多知识，并且可以搭建自己的网站。

14.2 Flask 框架应用基础

14.2.1 Flask 框架的安装与配置

目前 Linux 系统仍是大多服务器的解决方案，因此本章我们将在 Ubuntu 16.04 系统上演示 Flask 的安装与使用。

首先安装 virtualenv，virtualenv 会为应用创建一个独立的虚拟 Python 环境。

打开终端，输入如下命令开始安装：

```
sudo apt-get install python-virtualenv
```

virtualenv 安装完成后，在个人空间文件夹下创建并切换到新目录作为 Flask 项目的目录：

```
cd ~
mkdir mypoj
cd mypoj
```

在项目的目录下，执行如下代码创建并激活一个虚拟 Python 环境：

```
virtualenvenv
. env/bin/activate
```

创建虚拟环境的过程需要一定的时间，virtualenv 还会自动为新的 Python 虚拟环境安装 pip 等工具。当虚拟环境激活后，在终端命令行最前端会出现(env)字样提示这是虚拟环境，如图 14.1 所示。

```
ubuntu@ubuntu: ~/mypoj
ubuntu@ubuntu:~$ cd mypoj/
ubuntu@ubuntu:~/mypoj$ . env/bin/activate
(env) ubuntu@ubuntu:~/mypoj$
```

图 14.1　进入虚拟 Python 环境

下面在虚拟环境中安装 Flask，在虚拟环境下，在终端输入如下代码安装 Flask 与其依赖的其他组件：

```
pip install Flask
```

等待 pip 安装结束后，我们使用 Python 命令行测试 Flask 是否安装成功：

```
import flask
```

如果没有报错，则说明安装成功，如图 14.2 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>>
```

图 14.2　验证 Flask 模块安装

14.2.2　Flask 使用基础

本节中，我们会从零开始，一步一步编写并完善我们的基于 Flask 的 Web 项目，读者可以跟随本书，一点一滴地学习 Flask 框架使用的础知识。

1. 创建 Flask 项目

Web 项目逻辑往往比较复杂，无论是为了后期维护还是功能拓展，我们都需要认真考虑项目的目录结构。本章中，我们将使用一种较为清晰且常用于中小型项目的文件组织结构，读者可以参考本章中的文件目录结构，开发自己的 Flask 项目。

首先我们将目录切换到之前建立的 mypoj 目录中，并激活虚拟 Python 环境(参考上一节中的步骤)。在该目录下建立 app 目录，作为程序包目录。随后切换到 app 目录，新建 __init__.py 作为 Web 程序入口，在其中编写如下代码：

```
from flask import Flask

app = Flask(__name__)
```

这段代码很简单,首先导入 flask 模块,并创建 Flask 应用对象。

随后我们在 mypoj 目录下新建 run. py 脚本,用于调试项目:

```
#!env/bin/python

from app import app

app.run(debug = True)
```

保存后,我们需要为该脚本添加执行权限,在当前目录下执行如下 Linux Shell 脚本(不是 Python 脚本):

```
chmod a + x run.py
```

完毕后,我们只需调用如下 Shell 脚本即可调试我们的 Flask 项目:

```
./run.py
```

执行后,有如下的反馈,说明项目启动成功,如图 14.3 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ ./run.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 247-949-092
```

图 14.3 启动 Flask 服务器

2. 编写视图模块

虽然我们的项目可以启动了,但是还不能返回页面内容。下面我们就开始编写视图模块,来为 url 分配对应的页面视图。

在编写之前,我们首先需要了解 Flask 中路由的概念。在 Flask Web 项目中,当我们访问一个 url 时,Flask 会根据我们定义的路由为其分配对应的函数返回页面内容。我们可以通过 route 装饰器来定义路由规则。

我们在 app 目录下新建视图模块 view. py,编写如下代码:

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

这段代码使用了两个 route 修饰器，为访问“/”与“/index”的请求分配 index()函数为其返回页面内容。

随后在__init__.py 文件结尾继续添加如下代码，为其导入视图模块：

```
from app import view
```

保存后，我们使用上节中的 run.py 调试项目，在本机的浏览器中访问“http://localhost:5000/”或“http://localhost:5000/index”，可以看到返回的内容，如图 14.4 所示。

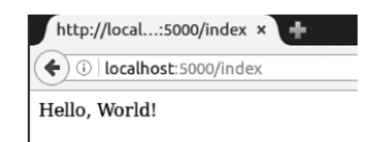


图 14.4 显示 Hello World

我们返回的内容也可以是 HTML 页面，我们修改 index()函数的返回值如下：

```
return"< html >< head ></head >< body >< font color = 'red'> Hello </font >, World! </body ></html >"
```

保存后重新运行项目，再次访问“http://localhost:5000/index”，得到如下页面，如图 14.5 所示。

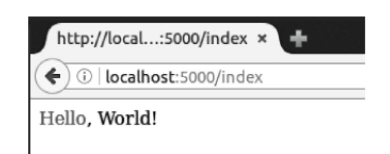


图 14.5 直接返回 HTML 代码

真实的网页往往包含很长的 HTML 代码，在 Python 内直接编写 HTML 的方式显然不可取。同时，动态网页中很多元素需要动态获取，这似乎又需要我们使用 Python 生成 HTML 代码。此时，应该怎么办呢？

还好 Flask 为我们整合了 jinja2 模板引擎，jinja2 模板允许开发者在 HTML 代码中嵌入特定的符号块，在渲染时只需要将这些动态符号块的内容作为参数传递即可。这样既满足了逻辑代码与 HTML 页面代码分离、将 HTML 代码单独存放在文件中的需求，也允许我们使用简洁的方法动态渲染 HTML 页面。下面我们将简单介绍 Flask 与 jinja2 的结合使用。

首先，我们需要在 app 目录下新建 templates 目录，该目录用于存放页面的模板；再在 app 目录下新建 static 目录用于存放 HTML 使用的静态资源如 js 文件、图片文件等。此时 app 的文件目录结构如图 14.6 所示。

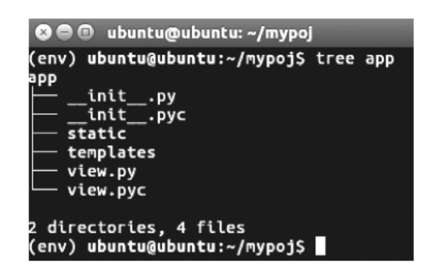


图 14.6 app 中的目录结构

在 app/templates 目录下，新建 index.html 文件，写入如下添加了 jinja2 语句块的代码（我们将稍后讲解 jinja2 语句块的语法）并保存：

```
<!DOCTYPE html>
<html>
<head>
        <title>Index Page</title>
</head>
<body>
        <p>Welcome,{{user['nickname']}}!</p>
        {% if user['age']>=18 %}
        <p>You are an adult!</p>
        {% else %}
        <p>You are not an adult!</p>
        {% endif %}
        <p>You like to eat:</p>
        <p>
                {% for item in fruit %}
                <b>{{item}}</b>,
                {% endfor %}
        </p>
</body>
</html>
```

随后，我们对 app/view.py 做如下修改(粗体部分)：

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
        return render_template("index.html",
                user = { 'nickname':'Tom', 'age':20 },
                fruit = ['apple','pear','orange'])
```

保存后重新运行项目，得到如图 14.7 所示的结果。

render_template 函数用于渲染 jinja2 模板，返回渲染后的 HTML 代码。其中第一个参数为模板名，即在 app/templates 目录下的某一模板的文件名；其他参数为 jinja2 模板中定义的变量。可见，我们返回的页面根据 user 和 fruit 进行了相应的变化，这便归功于 jinja2 模板。下面我们来介绍 jinja2 语句块的语法，分析 index 页面是如何被"生产"出来的。

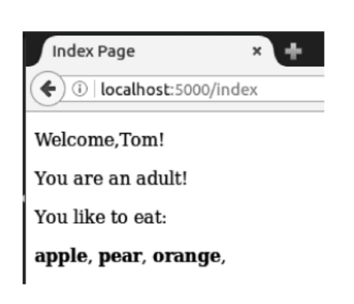


图 14.7　使用简单的 jinja2 模板

Jinja2 允许开发者在 HTML 代码中直接插入特定的语句块实现相应的效果，如我们的 index.html 中，大部分内容还是我们熟悉的 HTML 代码，只是其中穿插了{{...}}、{% ... %}等语句块，下面我们

来介绍 jinja2 常用的代码块。

3. 动态内容{{...}}

动态网页中一些内容往往需要根据不同场合改变,例如欢迎文本中的名字就需要根据实际用户名字进行改变,此时,我们只需要在动态文本的位置插入{{...}}块即可。在{{...}}块中,我们需要定义其展示的变量名。例如在 index.html 中,我们在 Welcome 后的{{...}}中展示了字典 user 的 nickname 索引下的内容:

```
<p>Welcome,{{user['nickname']}}!</p>
```

4. 条件控制{% if ... %}{% elif ... %}{% else %}{% endif %}

有时,页面需要根据一定条件进行改变,例如本例中如果 user 的 age 大于等于 18,我们会告知用户其为成年人,否则告知其不是成年人。我们只需把条件控制语句写在{% ... %}之间,满足某条件显示的内容写在相应的两个{% ... %}之间即可使用。需要注意的是,当{% if ... %}结束后,我们需要使用{% endif %}告知其 if 已经结束。

```
{% if user['age']>=18 %}
<p>You are an adult!</p>
{% else %}
<p>You are not an adult!</p>
{% endif %}
```

5. 循环语句{% for ... %}{% endfor %}

动态页面同样需要用循环语句进行控制。同样,我们只需将循环的条件写在{% ... %}之间,在循环结束后使用{% endfor %}告知其循环结束,并把循环展示的内容写在二者之间即可。

```
<p>
    {% for item in fruit %}
    <b>{{item}}</b>,
    {% endfor %}
</p>
```

Jinja2 模板将 Web 应用的逻辑代码与 HTML 页面很好地分离开又使其不失控制,使项目更加整洁美观。

不仅如此,jinja2 还支持模板的继承,我们可以把一个页面拆分成几个部分,使用继承的方法将各个部分连接起来,使得其他页面可以重用被拆分的部分,更加方便了页面代码的编写与维护,使得重复的内容只需要从中拆开写成一个单独的部分再用其他部分继承它即可。

6. 模板继承{% block ... %}{% endblock %}与{% extends ... %}

Jinja2 的继承方式也很简洁,我们只需要在被继承的页面中需要拓展的部分插入{% block ... %}{% endblock %},其中...为该被继承块的名称,因为一个页面可能存在多个块需要被继承,因此我们可以使用不同的名称进行区分。

我们将 index. html 改名为 base. html 并为其做如下修改（粗体）：

```
<!DOCTYPE html>
<html>
<head>
        <title>Index Page</title>
</head>
<body>
        <p>Welcome,{{user['nickname']}}!</p>
        {% if user['age']>=18 %}
        <p>You are an adult!</p>
        {% else %}
        <p>You are not an adult!</p>
        {% endif %}
        <p>You like to eat:</p>
        <p>
                {% for item in fruit %}
                <b>{{item}}</b>,
                {% endfor %}
        </p>
        {% block image %}{% endblock %}
</body>
</html>
```

随后在 app/templates 目录下重新建立 index. html 并写入如下内容：

```
{% extends "base.html" %}
{% block image %}
    <img src='static/python.png'/>
{% endblock %}
```

同时在 app/static 目录下放入一张名为 python. png 的图片。所有更新保存后，重新运行项目，得到如图 14. 8 所示的页面。

可见，index. html 成功继承了 base. html 中的内容并补充了一张图片。

7. 处理表单

在 Web 应用中，无论是登录、留言，往往都使用表单对内容进行封装。为了网站的安全性，防止 CSRF 攻击，我们将使用 Flask 的拓展模块 Flask-WTF，读者可以在 Linux Shell 下使用 pip 方便地安装这个模块：

```
pip install Flask-WTF
```

为了使我们的表单验证支持 CSRF 防御，我们需要为我们的 Flask 项目进行配置。出于安全性与日后的维护考虑，我们将配置单独写入一个文件并在 app/__init__. py 中导入该配置。首先，我们在项目根目录 mypoj 下新建 config. py：

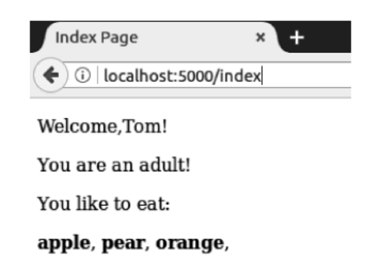




图 14.8 使用 jinja2 模板

```
CSRF_ENABLED = True
SECRET_KEY = "you-will-never-guess"
```

其中第一条配置用于开启 CSRF 防御,第二条代码用于配置 CSRF 防御使用的 secret_key,secret_key 应自定义为一个足够复杂难以被猜到的字符串(关于使用 secret_key 防御 CSRF 攻击原理读者可以在网上搜索相关资料,本书不深入讨论)。

编写好配置文件后,我们需要在 app/__init__.py 中导入配置,修改 app/__init__.py 如下(加粗部分):

```
from flask import Flask

app = Flask(__name__)
app.config.from_object('config')
from app import view
```

这样,我们便成功地对我们的项目进行了配置。下面来介绍表单的使用。

同样,为了使代码简洁便于维护,我们将所有的表单写入同一个文件中,新建 app/form.py,写入如下代码:

```
from flask_wtf import FlaskForm
from wtforms import StringField,PasswordField,BooleanField
from wtforms.validators import DataRequired,Length

class LoginForm(FlaskForm):
        nickname = StringField('nickname',
                validators=[DataRequired(message='nickname is required!'),
```

```
                    Length(6,20,message = 'nickname requires 6 - 20 characters! ')])
    password = PasswordField('password',
            validators = [DataRequired(message = 'password is required! '),
                    Length(6,20,message = 'password requires 6 - 20 characters! ')])
    remember_me = BooleanField('remember_me', default = False)
```

首先,我们从 flask. ext. wtf 中导入 FlaskForm 类,该类为我们自定义表单的基类,我们的自定义表单需要继承这个类;再从 wtforms 模块中导入三种表单元素,用来定义表单包含的元素类型;最后从 wtforms. validators 中导入两种验证器(我们将稍后介绍它们的作用)。

导入需要的内容后,我们创建我们的登录表单类 LoginForm,它要继承 FlaskForm 类。在 LoginForm 中,我们定义了三个表单元素,分别为 nickname、password 与 remember_me,对应为昵称、密码与"记住我"选项,其中前两个元素我们使用了验证器验证表单合法性。例如 nickname 元素,我们分别为其指定了 DataRequired 验证器与 Length 验证器。DataRequired 验证器用于必填元素,当该元素为空时,会提示 message 参数中的错误信息;Length 验证器用于验证内容长度,其前两个参数分别为内容的最小、最大字符数,同样,不满足条件时会提示 message 中的错误信息。

保存后,我们新建登录页面的模板,新建 app/templates/login. html:

```
<! DOCTYPE html >
< html >
< head >
        < title > Login Page </title >
</head >
< body >
        < form method = "POST">
        {{form. hidden_tag()}}
        < p >
                Please enter your nickname:< br/>
                {{form. nickname}}< br/>
                { % for e in form. nickname. errors % }
                < span style = 'color:red'> * {{e}}</span >
                { % endfor % }
        </p >
        < p >
                Please enter your password:< br/>
                {{form. password}}< br/>
                { % for e in form. password. errors % }
                < span style = 'color:red'> * {{e}}</span >
                { % endfor % }
        </p >
        < p >{{form. remember_me}}Remember Me </p >
        < p >< input type = "submit" value = "Sign In"></p >
        </form >
</body >
</html >
```

这段代码我们不再详细讲解,需要注意的是为了开启 CSRF 防御,我们需要在表单标签内嵌入{{form.hidden_tag()}};验证不通过的元素 message 在对应元素的 errors 列表中。

最后,我们要在视图模块中为登录页面注册路由,修改 app/view.py 如下(加粗部分):

```
from flask import render_template,url_for,redirect
from app import app
from forms import LoginForm

@app.route('/')
@app.route('/index')
def index():
        return render_template("index.html",
                user = { 'nickname':'Tom', 'age':20 },
                fruit = ['apple','pear','orange'])

@app.route('/login',methods = ['GET','POST'])
def login():
        form = LoginForm()
        if form.validate_on_submit():
                return redirect(url_for('success'))
        else:
                return render_template('login.html',form = form)

@app.route('/success')
def success():
        return '<h1>Success!</h1>'
```

除了 render_template 函数外,我们额外导入了 url_for 函数与 redirect 函数。redirect()函数用于 url 的重定向,使表单验证通过后跳转到成功页面;url_for()函数的作用是为函数生成 url 链接,例如这段代码中我们为 success()函数使用了 url_for 生成其对应的 url,得到 localhost:5000/success 的 url(调试环境下)。

这段代码还需关注的就是登录页面的路由与函数,登录页面的路由中,我们设置了 methods 参数,使其支持 GET 与 POST 两种提交方式。在 login()函数中,我们首先创建了登录表单的实例 form,然后判断 form 是否被正确填写,如果添加了验证器的元素满足对应需求,则将链接重定向到 success 页面,否则渲染 login.html 页面,其中模板中的 form 参数使用我们实例化的登录表单填充。

重新运行项目,访问登录页面,如图 14.9 所示。

我们测试不符合要求的数据,如图 14.10 所示。

提交后,得到如图 14.11 所示的提示信息。

修改登录信息,如果符合要求,链接将被重定向到 success 页面,如图 14.12 所示。

8. 使用数据库

在之前的章节中,我们介绍了数据库的使用。使用数据库可以方便地对数据进行管理与查询。在开发 Flask 应用时,同样可以使用我们之前章节中介绍的操作数据库的方法。当然,Flask 同样提供了一个基于 ORM 的数据库拓展模块:Flask-SQLAlchemy,它支持多

种数据库，并且支持 Flask-Migrate 模块方便的版本控制与数据库迁移 API，让用户遇到必要的变更与升级时可以快速更新数据库结构而不需要新建数据库并手动迁移数据。下面我们就来介绍 Flask-SQLAlchemy 与 Flask-Migrate 的安装与使用。

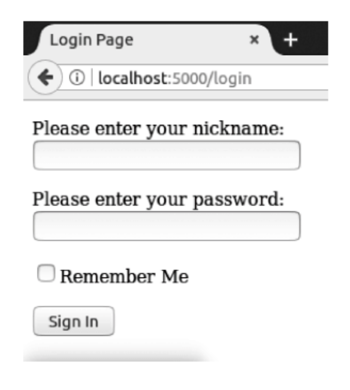


图 14.9　登录页面

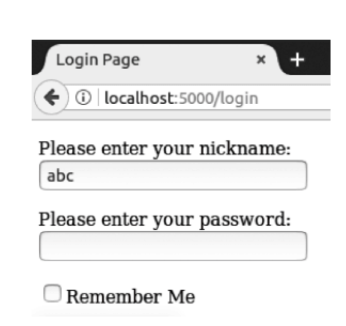


图 14.10　测试不符合要求的数据

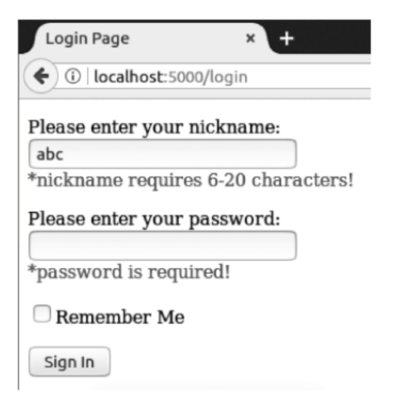


图 14.11　错误信息

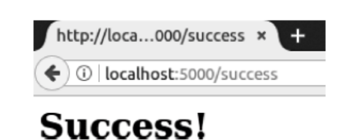


图 14.12　登录成功页面

Flask-SQLAlchemy 与 SQLAlchemy-migrate 是 Flask 的一个拓展模块。我们仍需要使用 pip 安装，其中 Flask-Migrate 还需要 Flask-Script 等依赖，pip 会自动安装依赖：

```
pip install Flask-SQLAlchemy
pip install Flask-Migrate
```

安装完成后，我们需要对 SQLAlchemy 模块进行配置，修改项目根目录 mypoj 下的 config.py 如下（加粗部分）：

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))

SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db')
SQLALCHEMY_TRACK_MODIFICATIONS = True
```

```
CSRF_ENABLED = True
SECRET_KEY = "you-will-never-guess"
```

新增的配置中 SQLALCHEMY_DATABASE_URI 为 Flask-SQLAlchemy 需要的配置,它的值为我们的数据库类型与路径(本文中我们使用 sqlite 数据库);SQLALCHEMY_TRACK_MODIFICATIONS 用于开启或关闭数据库操作的回显,在调试项目时,我们可以开启这个选项用于观察数据库的变化。

修改配置文件完毕后,我们需要在 app/__init__.py 中实例化数据库对象,修改 app/__init__.py 如下(加粗部分):

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

from app import view,models
```

在新增的代码中,我们导入了 Flask-SQLAlchemy 拓展的 SQLAlchemy 类,并初始化其数据库实例 db。同时我们导入了 models 模块。models 模块是我们自定义的 ORM 模型模块,下面我们将编写我们的 models。

在之前的章节中,我们介绍过 ORM 数据库模型的使用。在 Flask-SQLAlchemy 中,使用方法与以前几乎没有区别,我们不再详细介绍 ORM 的使用,请读者根据模型的代码自行理解 ORM 的映射关系,理解困难的读者可以参考之前的章节,下面我们编写 app/models.py 如下:

```
from app import db
import hashlib

def getSHA256(content):
        SHA256 = hashlib.sha256()
        SHA256.update(content)
        return SHA256.hexdigest()

class User(db.Model):
        id = db.Column(db.Integer, primary_key = True)
        nickname = db.Column(db.String(64), index = True, unique = True)
        password = db.Column(db.String(64))

        def __init__(self,_nickname,_password):
                self.nickname = _nickname
                self.password = getSHA256(_password)
```

```
        def __repr__(self):
                return '<User %r %r>' % (self.nickname,self.password)

        def getNickname(self):
                return self.nickname

        def getPassword(self):
                return self.password
```

这段代码中需要注意的是加粗部分，我们导入了 hashlib 模块，并使用其中的 SHA256 方式对 password 加密后再作为 password 字段的内容存储。在实际的网站开发中，我们应当为用户的密码进行散列加密而不是直接存储明文密码。这样，在网站数据库泄露后，黑客无法直接使用加密过的密码登录，而需要先对其进行解密，而散列算法的解密代价是巨大的。而且同一用户在不同网站可能会使用相同密码，存储加密后的密码是对用户负责的基本素养。在过去，大多数网站使用 md5 的方式对密码进行加密，但是随着 md5 彩虹表的完善与可撞库的逐渐增多，目前大多数常用密码的 md5 密文都可以以很低的价格破解。因此，本例中我们使用了目前更为安全的 SHA256 算法。对这方面内容感兴趣的读者可以深入学习。

现在我们有了数据库实例与 ORM 的映射关系，但是还没有创建数据库与迁移仓库。这时，我们就需要使用 Flask-Migrate 拓展。我们在项目的根目录 mypoj 下新建 manage.py 作为项目的管理脚本：

```
#!env/bin/python

from app import app,db
from flask_migrate import Migrate,MigrateCommand
from flask_script import Manager

manager = Manager(app)
migrate = Migrate(app,db)
manager.add_command('db',MigrateCommand)

manager.run()
```

这段代码中，我们从 app 中导入了应用程序 app 与数据库实例 db，并且导入了 Flask-Migrate 中的 Migrate 与 MigrateCommand 以及 Flask-Migrate 的依赖模块 Flask-Script 中的 Manager。我们为 app 创建了管理器 manager，并创建数据库迁移器 migrate，随后我们对管理器 manager 使用 add_command 方法，使我们可以在 Linux Shell 中直接使用脚本操作，更简化的操作，我们可以为其添加"db"命令，并指定"db"后的命令对应迁移命令 MigrateCommand。

同样，为了方便使用该脚本，我们使用 chmod 命令为其添加可执行权限，之后便可以使用./执行：

```
chmod a + x manage.py
```


一切准备就绪后，可以开始创建我们的数据库文件。

首先，我们需要使用 init 命令创建迁移仓库：

```
./manage.py db init
```

执行后，Flask-Migrate 会为我们创建 migrations 目录，其中包括数据库的版本信息与迁移脚本，随后我们可以进行第一次迁移：

```
./manage.py db migrate
```

因为我们之前并没有手动创建数据库，因此第一次迁移会首先根据我们的配置文件创建 SQLite 的数据库文件 app.db。执行 migrate 命令时，Flask-Migrate 会探测 app/models.py 中改变的模型并记录。注意，此时我们的数据库还未升级，migrate 仅仅会记下模型的变更，为升级数据库做准备。如图 14.13 所示，migrate 命令检测到了 user 表以及其索引信息。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ ./manage.py db init
  Creating directory /home/ubuntu/mypoj/migrations ... done
  Creating directory /home/ubuntu/mypoj/migrations/versions ... done
  Generating /home/ubuntu/mypoj/migrations/env.py ... done
  Generating /home/ubuntu/mypoj/migrations/README ... done
  Generating /home/ubuntu/mypoj/migrations/env.pyc ... done
  Generating /home/ubuntu/mypoj/migrations/script.py.mako ... done
  Generating /home/ubuntu/mypoj/migrations/alembic.ini ... done
  Please edit configuration/connection/logging settings in '/home/ubuntu/mypoj/migrations/alembic.ini' before proceeding.
(env) ubuntu@ubuntu:~/mypoj$ ./manage.py db migrate
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_nickname' on '['nickname']'
  Generating /home/ubuntu/mypoj/migrations/versions/d7cd0dad9906_.py ... done
(env) ubuntu@ubuntu:~/mypoj$
```

图 14.13　使用 Flask-Migrate 创建迁移脚本库

此时，如果我们使用 show 命令便可以查询到本次迁移，如图 14.14 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ ./manage.py db show
Rev: 1f3441fa6b45 (head)
Parent: <base>
Path: /home/ubuntu/mypoj/migrations/versions/1f3441fa6b45_.py

    empty message

    Revision ID: 1f3441fa6b45
    Revises:
    Create Date: 2017-04-08 01:39:43.991351

(env) ubuntu@ubuntu:~/mypoj$
```

图 14.14　查询迁移

在 migrate 后，便可以执行命令升级数据库：

```
./manage.py db upgrade
```

此时,我们可以使用 sqlite3 查看 app.db 中表的结构,如图 14.15 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ sqlite3 app.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT * FROM sqlite_master WHERE type='table';
table|alembic_version|alembic_version|2|CREATE TABLE alembic_version (
        version_num VARCHAR(32) NOT NULL,
        CONSTRAINT alembic_version_pkc PRIMARY KEY (version_num)
)
table|user|user|4|CREATE TABLE user (
        id INTEGER NOT NULL,
        nickname VARCHAR(64),
        password VARCHAR(64),
        PRIMARY KEY (id)
)
sqlite> .quit
(env) ubuntu@ubuntu:~/mypoj$
```

图 14.15 升级数据库

可见其中包含两个表,一个便是我们在模型中定义的 user 表。除此之外,数据库中还有 alembic_version 表,这个表表示 Flask-Migrate 为我们自动创建的,用于存放版本控制信息。

如果以后需要修改数据库模型,我们只需修改 app/models.py 并执行数据库迁移管理脚本的 migrate 与 upgrade 命令即可,不需要手动新建表并导入数据,更加方便管理与维护。

在编写完数据库管理脚本并建立了数据库后,我们便可以使用这个数据库了。

本节我们仅通过命令行的方式导入 db 进行测试,对于其在 Flask 应用中的使用,我们将在下一节介绍。

首先,启动 Python 命令行并从 app 中导入数据库对象 db,从 app.models 模块下导入 User 类,如图 14.16 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from app import db
>>> from app.models import User
>>>
```

图 14.16 在命令行中导入 db 与 User 类

我们新建 User 类的实例 u,并设置它的 nickname 为'Tom',设置 password 为'123456',如图 14.17 所示。

```
ubuntu@ubuntu: ~/mypoj
(env) ubuntu@ubuntu:~/mypoj$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from app import db
>>> from app.models import User
>>>
>>> u = User('Tom','123456')
>>> u
<User 'Tom' '8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92'>
>>>
```

图 14.17 创建新用户

可以看到，u 的 password 已经是使用 SHA256 加密后的密文。

随后，我们测试将 u 写入数据库，如图 14.18 所示（对操作不熟悉的读者可参考之前的章节，其中有专门对 ORM 数据库操作的讲解，这里大同小异）。

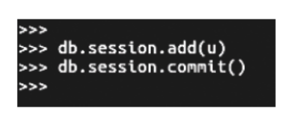


图 14.18　添加新用户

我们再测试查询数据库 User 表中的信息，如图 14.19 所示。

可见，我们已经成功将昵称为“Tom”的用户写入了数据库。

```
>>>
>>> users = User.query.all()
>>> users
[<User u'Tom' u'8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92'>]
>>>
```

图 14.19　查询用户

9. 用户登录、登出

无论是论坛还是电商网站，用户都需要登录才能使用一些特有的功能。Flask 的拓展 Flask-Login 模块为开发者提供了方便的用户登录的支持。

首先，我们使用 pip 安装 Flask-Login 拓展：

```
pip install Flask-Login
```

安装后，我们需要对 app/__init__.py 做如下修改：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)
lm = LoginManager(app)

from app import view,models
```

这段代码从 Flask-Login 中导入了登录管理器 LoginManager 并实例化为 lm。

为了使用 Flask-Login 模块，我们还需要对之前的 User 类进行如下修改：

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    nickname = db.Column(db.String(64), index = True, unique = True)
    password = db.Column(db.String(64))

    def __init__(self,_nickname,_password):
        self.nickname = _nickname
```

```
        self.password = getSHA256(_password)

    def __repr__(self):
        return '<User %r %r>' % (self.nickname,self.password)

    def getNickname(self):
        return self.nickname

    def getPassword(self):
        return self.password

    def is_authenticated(self):
        return True

    def is_active(self):
        return True

    def is_anonymous(self):
        return False

    def get_id(self):
        return unicode(self.id)
```

对 app/models.py 中的 User 类，我们为其添加了 4 个 Flask-Login 模块需要的方法：is_authenticated(self)方法用于返回用户认证状态，只有该函数返回 True 的用户才可以具有全部登录后可访问页面的访问权限，这个参数一般被置为 True；is_active(self)方法用于返回用户可用状态，可用返回 True，被停用返回 False；is_anonymous(self)方法用于该用户是否为匿名，如果是匿名用户返回 True，真实用户返回 False；前三个方法为用户的状态提供了多元的选择，不过本例中不需要这些功能，因此直接返回 True 或者 False，读者可以根据需求自定义实现；最后一个方法 get_id(self)用于返回用户唯一认证 id，本例中我们返回了被设置为 primary_key 的 id 字段。

为我们的 User 类实现这 4 个方法后，我们便可以在 Flask-Login 中使用 User 类作为用户类了。下面我们将在视图模块中进行修改，实现用户的登录、登出并设置登录后才能访问的页面。

我们对 app/view.py 进行如下修改（加粗部分）：

```
from flask import render_template,url_for,redirect,flash
from app import app,lm
from app.models import User,getSHA256
from forms import LoginForm
from flask_login import login_user,logout_user,login_required

lm.login_view = 'login'
```

```
@lm.user_loader
def load_user(id):
        return User.query.get(int(id))

@app.route('/')
@app.route('/index')
def index():
        return render_template("index.html",
                user = { 'nickname':'Tom' , 'age':20 },
                fruit = [ 'apple','pear','orange'])

@app.route('/login',methods = [ 'GET','POST'])
def login():
        form = LoginForm()
        if form.validate_on_submit():
                u = User.query.filter_by(nickname = form.nickname.data).first()
                if u is not None:
                        if u.getPassword() == getSHA256(form.password.data):
                                login_user(u,form.remember_me.data)
                                return redirect(url_for('success'))
                        else:
                                flash('Wrong password! ')
                else:
                        flash('Nickname not found! ')
        return render_template('login.html',form = form)

@app.route('/success')
@login_required
def success():
        return render_template('success.html')

@app.route('/logout')
@login_required
def logout():
        logout_user()
        return redirect(url_for('login'))
```

为了实现整套登录功能，本次的修改比较多，下面我们将一一讲解所有修改。

首先我们从 flask 模块中导入了 flash 函数，该函数用于为页面发送即时消息，接收即时消息的页面需要做一些修改来展示 flash 的消息，具体我们将在下面修改登录成功页面时介绍。我们还从 app 中导入了登录管理器 lm，从 app.models 中导入了 User 类与 getSHA256 方法用于验证用户身份，并从 Flask-Login 模块中导入了 login_user，logout_user，login_required，这些将在下文中详细介绍。

导入需要使用的功能后，我们首先修改登录管理器的 login_view 属性，该属性用来指定未登录的用户在访问只有登录过的用户才有权限访问的页面时被重定向到的路由函数

名，在本例中，登录用的路由的函数为 login。

Flask-Login 模块还需要我们实现 load_user 方法，使用修饰器@lm. user_loader 来指定，load_user 方法需要接收一个 id 参数，并根据这个唯一的 id 返回查询到的 User 类的实例，这个 id 也就是我们 User 类中 get_id 对应的唯一 id。注意，查询时 id 只接收整型参数，因此我们有必要将其转换为 int 类型。

对于用户身份的认证主要在 login 路由函数中实现。当我们接收的表单合法时，根据表单中的 nickname 元素查询数据库，因为 nickname 列是 unique 的，因此我们只需要使用 first()返回第一个查询到的 User 实例即可；如果数据库中没有 nickname 为表单中填写的 nickname 的行时，将返回 None。因此，我们作如下判断，如果 u 是 None，说明不存在该 nickname，我们使用 flash 返回不存在用户名的信息，否则验证表单中提交的密码。验证表单中提交的密码时，我们需要对其使用 SHA256 算法加密，将密文与根据 nickname 查询到的 User 实例 u 的 password 属性比较，如果不同则为 flash 密码错误的消息，否则说明用户认证成功。

对于认证成功的用户，我们使用之前导入的 login_user 方法将其交给登录管理器处理，login_user 还可以接收一个布尔参数表示是否记住登录，本例中我们传入了表单中的 remember_me 元素的值。登录管理器自动处理 cookie、session 等信息，十分便捷。登录成功后，我们便可以将连接重定向到 success 页面。

为了正确展示登录时 flash 的消息，我们对 app/templates/login. html 做如下修改（加粗部分）：

```
<!DOCTYPE html>
<html>
<head>
        <title>Login Page</title>
</head>
<body>
{% for message in get_flashed_messages() %}
                <span style='color:red'>{{ message }}</span>
{% endfor %}
        <form method="POST">
        {{form.hidden_tag()}}
        <p>
                Please enter your nickname:<br/>
                {{form.nickname}}<br/>
                {% for e in form.nickname.errors %}
                <span style='color:red'>*{{e}}</span>
                {% endfor %}
        </p>
        <p>
                Please enter your password:<br/>
                {{form.password}}<br/>
                {% for e in form.password.errors %}
```

```
                    <span style='color:red'>*{{e}}</span>
                    {% endfor %}
            </p>
            <p>{{form.remember_me}}Remember Me</p>
            <p><input type="submit" value="Sign In"></p>
        </form>
</body>
</html>
```

get_flashed_messages()会返回 flash 的消息列表，我们使用循环展示全部 flash 的消息即可。

为了测试登录功能，我们对 success 页面也做了一定的修改。首先，我们为其添加了@login_required 修饰器。添加了@login_required 修饰器的连接只能被已登录的用户正常访问，否则将会被重定位到登录管理器的 login_view 指定的页面。然后，我们编写了 success 页面的模板：

```
<!DOCTYPE html>
<html>
<head>
            <title>Login Page</title>
</head>
<body>
        <p>
            Welcome,{{current_user.nickname}}!
        </p>
</body>
</html>
```

在 success.html 中，我们在 jinja2 块中使用了 current_user，jinja2 会自动为这个变量传入我们已登录的用户对象，因此我们可以直接使用 current_user 来访问当前登录的用户信息。

除此之外，我们还添加了同样具有@login_required 修饰器的 logout 路由用于用户登出，登出时，只需调用 logout_user 函数即可。

在测试功能之前，我们先使用命令行为数据库中添加一个合法的用户（当然，读者可以再编写一个用户注册页面并在对应路由下操作数据库添加新用户），如图 14.20 所示。

启动项目，我们先不进行登录，直接访问 success 页面，连接被重定向到 login 页面，页面行还有一条自动的 flash 提示信息，如图 14.21 所示。

接下来我们尝试用错误的 nickname 或密码登录，观察 flash 消息的变化，如图 14.22 和图 14.23 所示。

接下来我们使用正确的信息登录，登录后成功跳转到了修改后的 success 页面，该页面通过 current_user 展示了我们登录的用户的 nickname，如图 14.24 所示。

接下来我们手动访问 logout 页面，连接会被重定向到 login 页面（此处不进行演示）。在登出之后，如果我们再次访问 seccess 页面，访问将再次被拒绝并跳转到 login 页面且显示提示信息，如图 14.25 所示。

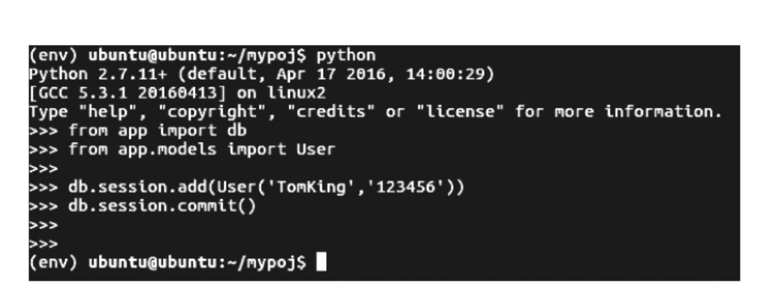


图 14.20　添加测试用户

Login Page
localhost:5000/login?next=%2Fsuccess
Please log in to access this page.
Please enter your nickname:
Please enter your password:
Remember Me
Sign In

图 14.21　登录保护

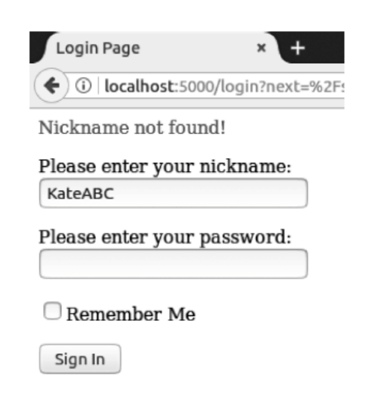


图 14.22　未找到用户的报错信息

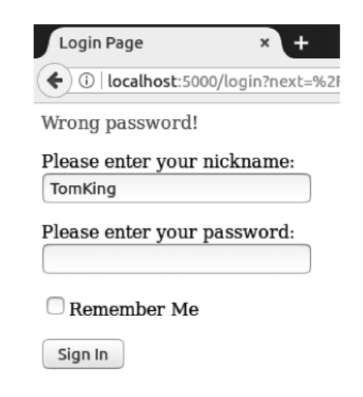


图 14.23　密码错误的报错信息



图 14.24　登录成功

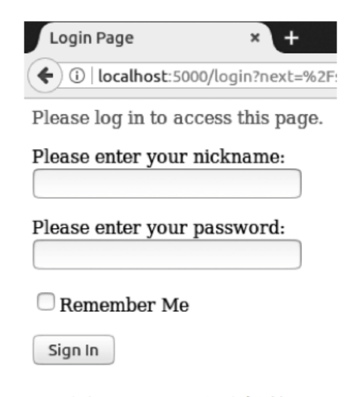


图 14.25　登录保护

关于用户登录，我们就介绍到这里，读者可以根据自己的需求实现更多功能。

14.2.3　在服务器上部署 Flask 项目

对于 Flask 的开发部分我们就简单介绍到这里。Web 是一个十分广泛的话题，关于 Web 应用开发的知识与经验也一言难尽，对这方面感兴趣的读者可以学习这方面的专著。

之前我们一直在使用 Debug 的模式运行我们的 Flask 应用并通过本地 5000 端口访问，在实际环境下，我们需要使用服务器应用在 80 端口下部署我们的应用，才能在互联网中正常使用。虽然 Flask 自带的 WSGI 服务器框架可以方便地让我们部署项目，但是考虑性能因素，我们还是需要更加强大的服务器应用来在真实的网络环境下运行我们的 Flask 项目。

下面我们为读者介绍使用支持反向代理的 nginx 服务并结合 uWSGI 来部署我们的 Flask 应用，使其可以在公网上的主机上稳定地运行。

首先，我们需要在 Ubuntu 上使用 apt 安装 nginx 与 python-dev：

```
sudo apt - get install nginx
sudo apt - get install python - dev
```

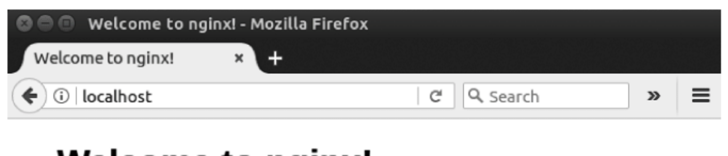
接下来，我们还需要使用 pip 安装 uWSGI：

```
pip install uwsgi
```

安装成功后，我们先来测试 nginx 是否可用，在 Linux Shell 中执行下面这段代码：

```
sudo /etc/init.d/nginx start
```

接下来，我们在服务器上直接访问 localhost 或者在其他与该服务器建立了连接的主机访问该服务器地址即可看到 ngxin 的欢迎信息，说明 nginx 安装成功并能够正常运行，如图 14.26 所示。



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

图 14.26　验证 nginx 安装

下面需要将我们的项目拷贝到/var/www 下进行托管。

首先执行命令：

```
cd ~
sudo cp - r mypoj/ /vai/www/mypoj
```

这段命令将我们的 mypoj 目录使用管理员权限拷贝到了/var/www 目录下。此时，/var/www/下的 mypoj 目录及其中文件属于 root 所有，我们需要再次使用管理员权限将其

所有者更改为我们当前的用户(以 ubuntu 为例):

```
sudo chown -R ubuntu:ubuntu /var/www/mypoj/
```

接下来,我们还需要修改 run.py 脚本,使其可以在真实环境中使用。我们先复制一个运行脚本并命名为 run-debug.py:

```
cd /var/www/mypoj
cp run.py run-debug.py
```

再在 run.py 中进行修改:

```
#!env/bin/python

from app import app

if __name__ == "__main__":
        app.run(host='0.0.0.0',port=5000)
```

run 函数的 host 参数如果指定为"0.0.0.0"即允许 Flask 监听端口下所有连接的请求。运行 run.py 让我们的 Flask 应用在 5000 端口正常启动,访问本地 5000 端口测试,出现我们最开始编写的 index 页面,如图 14.27 所示。

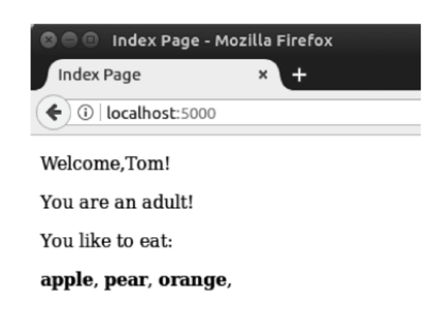




图 14.27　监听所有 IP 地址

此时,Flask 仍然在由其内置的 Web 服务托管,下面修改 nginx 的配置,使用 nginx 托管我们的项目。

首先,删除 Nginx 的默认配置文件:

```
sudo rm /etc/nginx/sites-enabled/default
```

接着，在 mypoj 目录下建立我们 Flask 应用专用的配置文件/var/www/mypoj/nginx.conf：

```
server {
    listen      80;
    server_name localhost;
    charset     utf-8;
    client_max_body_size 75M;

    location / { try_files $uri @app; }
    location @app {
        include uwsgi_params;
        uwsgi_pass unix:/var/www/mypoj/uwsgi.sock;
    }
}
```

以上配置确定了监听的端口、服务器、字符集等信息以及使用 uWSGI 的 socket 文件路径，下面来配置 uWSGI：

新建 uWSGI 配置文件/var/www/mypoj/uwsgi.ini：

```
[uwsgi]
#application's base folder
base = /var/www/mypoj

#python module to import
app = app
module = %(app)

home = %(base)/env
pythonpath = %(base)

#socket file's location
socket = %(base)/uwsgi.sock

#permissions for the socket file
chmod-socket = 666

#the variable that holds a flask application inside the module imported at line #6
callable = app

#location of log files
logto = %(base)/uwsgi.log
```

保存配置文件后，我们只需要应用 uwsgi 配置文件启动 uwsgi 服务即可：

```
uwsgi --ini /var/www/mypoj/uwsgi.ini
```

现在直接访问 localhost，出现的便是我们的 index 页面，如图 14.28 所示。

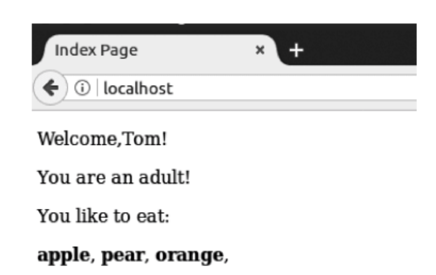




图 14.28　使用 nginx 代理端口

至此，我们便使用 nginx、uWSGI 成功地部署了我们的项目。

习　题　14

一、填空题

1. 在 Linux 下，我们可以使用________工具来创建虚拟 Python 环境。

2. 以调适模式启动 Flask 服务器的命令是________。

3. Flask 的路由默认使用________方式，修改________的________参数可以使其接受“POST”方式的访问。

4. 为了安全与便捷，我们可以使用________模块来处理表单。如果开启了 CSRF 防御，我们需要在 HTML 页面的表单中添加________，并在配置文件中配置________。

5. Flask 提供了________模块作为 ORM 工具。

二、论述题

1. 简述使用散列算法（如 SHA256、MD5、SHA1 等）对密码加密后保存的意义。

2. 查阅相关资料，了解 Flask 模块其他拓展的功能以及使用方法。

三、编程题

使用 Flask 框架实现自己的博客网站。

第 15 章　Python 综合应用实例

通过之前章节的学习，读者已经掌握了 Python 编程基础的大部分内容。在本章中，我们将综合之前学习的内容，将学到的 Python 应用到实际开发中。通过一些简单的例子，回顾之前的内容，提高读者使用 Python 开发的能力。

15.1　带图形界面的简易计算器

知识点：Tkinter 图形界面编程、匿名函数的应用、简单的异常处理。

在制作简易的图形界面计算器之前，先思考一下我们制作的计算器需要哪些功能与组件。首先，我们需要按下按钮输入表达式，表达式需要支持加、减、乘、除、括号、取模、数字和小数点；其次，我们还需要一个按钮来清空表达式；接下来，我们需要“等号”按钮计算表达式；最后我们需要显示表达式与结果。显示器我们使用 Tkinter 的 Entry 输入框来实现，其他按钮我们使用 Button 实现。

在编译类的语言中，实现表达式求值不是一件容易的事，需要结合数据结构“栈”来运算。不过 Python 作为一种动态的解释型语言，我们可以使用稍微“偷懒”的方法，即使用“eval(表达式)”函数来计算表达式的值。不过，在真正开发项目时，不建议使用 eval 函数，因为 eval 函数会导致任意代码执行漏洞，影响安全性。

首先，我们将计算器的界面封装到 App 类中进行编写：

```
# -*- coding:utf-8 -*-
import Tkinter as tk


class App:
    def __init__(self, master):
        self.master = master
        frame = tk.Frame(master)
        frame.pack()
        keys = '789+456-123*0./%()'
        s = tk.StringVar()
        screen = tk.Entry(frame, textvariable=s, state='readonly')
        screen.grid(column=0, row=0, columnspan=4)

        for x in range(0, 4):
            for y in range(1, 6):
```

```
                if x + (y - 1) * 4 >= len(keys):
                    break
                button_key = keys[x + (y - 1) * 4]
                tk.Button(frame, text = button_key, width = 3).grid(column = x, row = y)
        tk.Button(frame, text = 'C', width = 3).grid(column = 2, row = 5)
        tk.Button(frame, text = '=', width = 3).grid(column = 3, row = 5)


if __name__ == '__main__':
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```

运行后，我们得到如图 15.1 所示的界面。

为了在使用 eval 函数的同时尽可能提高安全性，我们将输入框修改为只读模式，这样就只能通过单击按钮来输入表达式。下面我们需要为按钮添加命令。

首先，数字、运算符、括号、小数点对应的按钮的功能都是相似的。用户按下这些按钮时，计算器的显示器上应该追加对应的字符。这里，我们使用 Button 组件的 command 属性和 lambda 匿名函数的方式来实现，将创建这些按钮的代码按照如下的方式修改：

```
tk.Button(frame, text = button_key, width = 3,
  command = lambda s = s, c = button_key: s.set(s.get() + c)
).grid(column = x, row = y)
```

在这段代码中，我们使用 lambda 创建了匿名函数，传入了显示器的字符串变量与按钮的字符，我们在显示器字符串变量后追加了按钮对应的字符。这样即可实现单击按钮输入表达式的功能。依次单击“1”“+”“1”后，得到如图 15.2 所示的效果。

图 15.1　计算器界面

图 15.2　单击按钮输入表达式

接下来我们编写清空表达式的命令。这个命令同样使用匿名函数实现，我们将按钮“C”的代码修改如下：

```
tk.Button(frame, text = 'C', width = 3, command = lambda s = s: s.set("")).grid(column = 2, row = 5)
```

最后需要编写计算表达式并显示的函数。因为在这个过程中，我们需要使用异常处理

来处理错误的输入，所以需要编写一个函数“cal”用来计算。因为“cal”函数中需要传入显示器的字符串变量，因此我们需要使用匿名函数结合一般函数的方法来实现，添加了所有功能的代码如下：

```
# -*- coding:utf-8 -*-
import Tkinter as tk


def cal(s):
    try:
        s.set(eval(s.get()))
    except:
        s.set("Error!")


class App:
    def __init__(self, master):
        self.master = master
        frame = tk.Frame(master)
        frame.pack()
        keys = '789+456-123*0./%()'
        s = tk.StringVar()
        screen = tk.Entry(frame, textvariable=s, state='readonly')
        screen.grid(column=0, row=0, columnspan=4)

        for x in range(0, 4):
            for y in range(1, 6):
                if x + (y - 1) * 4 >= len(keys):
                    break
                button_key = keys[x + (y - 1) * 4]
                tk.Button(frame, text=button_key, width=3,
                    command=lambda s=s, c=button_key: s.set(s.get() + c)
                ).grid(column=x, row=y)
        tk.Button(frame, text='C', width=3,
                command=lambda s=s: s.set("")).grid(column=2, row=5)
        tk.Button(frame, text='=', width=3,
                command=lambda s=s: cal(s)).grid(column=3, row=5)


if __name__ == '__main__':
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```

我们在 cal 函数中使用 try/except 处理异常，如果表达式错误，显示“Error!”

接下来，我们运行并测试这个计算器。首先，计算表达式“(1+2)*4/3”，得到的结果如图 15.3 所示。

接下来输入错误的表达式“2.*/0”，计算后会显示如图 15.4 的错误信息。



图 15.3　计算表达式

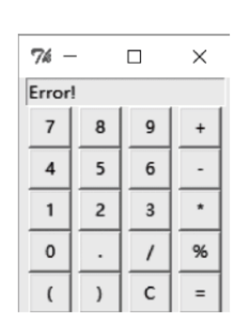


图 15.4　表达式错误

15.2　简单的网络爬虫

知识点：网络请求、正则表达式、数据库应用、多线程编程。

网络爬虫是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本。通过网络爬虫，我们可以快速获取网络上的信息，便于之后的统计分析与调查研究。本节我们将以爬取豆瓣电影 TOP250 为例，为读者介绍简单的爬虫技术。

在我们通过浏览器输入 HTTP 协议的网址访问网页时，我们实际上是向对应的服务器发送了一个 GET 请求。HTTP 协议规定了很多种请求，如 get、post、put、delete 等。网站服务器的后端程序从数据库提取数据、渲染模板后，将 HTML 网页发送回访问者，访问者的浏览器会对 HTML 渲染，最终呈现给用户。我们的简单的爬虫同样采取这种策略：发送 get 请求、从返回的 HTML 页面中提取信息、存储信息。

首先，我们需要解决发送请求的问题。Python 提供了 urllib、urllib2 库来支持 Web 访问。不过这两个库提供的功能比较全面，操作相对复杂。我们可以使用 Python 的第三方库 requests 来操作 Web 访问，可以使用 pip 安装 requests 模块：

```
pip install requests
```

安装成功后，我们先模拟发送一次 get 请求：

```
# - * - coding:utf - 8 - * -
import requests

url = 'https://movie.douban.com/top250'
headers = {
    'User - Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36',
}
res = requests.get(url, headers = headers)
print res.text
```

在这段代码中，我们使用了 requests 的 get 方法来发送 get 请求，get 方法的第一个参数是访问的 url，我们还自选了 headers 参数。headers 参数用来自定义请求的 Header 头信

息，headers 是一个字典类型。在本例中，我们自定义了 User-Agent 参数，这个参数一般用来标识发送请求的环境。因为很多网站会通过 User-Agent 检测爬虫程序，因此我们需要模拟浏览器的 User-Agent 来发送请求。requests 的 get 方法会返回一个 Response 对象，其有 text 属性，这个属性即后端返回的 HTML 页面代码。读者运行这段代码后，可以输出豆瓣电影 TOP250 的 HTML 页面代码。

接下来需要从这个 HTML 页面中提取有用的信息。我们在浏览器中访问这个页面，可以看到网页中罗列了电影列表，如图 15.5 所示。在列表中，只提供了很少的电影相关的信息。因此，如果想获取更多的信息，我们需要访问电影的详细页。因此，我们需要从列表中获取电影详细页的 URL 地址。

豆瓣电影 Top 250

我没看过的

1 肖申克的救赎 / The Shawshank Redemption / 月黑高飞(港) / 刺激1995(台) [可播放]

导演: 弗兰克·德拉邦特 Frank Darabont 主演: 蒂姆·罗宾斯 Tim Robbins /...

1994 / 美国 / 犯罪 剧情

9.6 871743人评价

希望让人自由。

2 霸王别姬 / 再见，我的妾 / Farewell My Concubine [可播放]

导演: 陈凯歌 Kaige Chen 主演: 张国荣 Leslie Cheung / 张丰毅 Fengyi Zha...

1993 / 中国大陆 香港 / 剧情 爱情 同性

9.5 626475人评价

风华绝代。

3 这个杀手不太冷 / Léon / 杀手莱昂 / 终极追杀令(台) [可播放]

导演: 吕克·贝松 Luc Besson 主演: 让·雷诺 Jean Reno / 娜塔莉·波特曼 ...

1994 / 法国 / 剧情 动作 犯罪

9.4 833747人评价

怪蜀黍和小萝莉不得不说的故事。

图 15.5 豆瓣电影 TOP250 页面

在浏览器中按 F12 键，可以打开浏览器的调试面板。我们以 Chrome 浏览器为例，在打开的 Elements 面板中，我们可以快速访问 HTML 代码内容，从中可以看到电影的详细页 URL 如图 15.6 所示。

```
▼<div class="info">
  ▼<div class="hd">
    ▼<a href="https://movie.douban.com/subject/1292052/" class>
        <span class="title">肖申克的救赎</span>
        <span class="title"> / The Shawshank Redemption</span>
        <span class="other"> / 月黑高飞(港)  /  刺激1995(台)</span>
      </a>
      <span class="playable">[可播放]</span>
```

图 15.6 电影详细页的 URL

我们需要使用正则表达式从中提取出电影的 URL 连接。另外，豆瓣电影 TOP250 每页只显示 25 条信息，我们需要分 10 次爬取列表才能获取完整的 250 条电影列表，在浏览器中单击第二页，查看 URL 的变化，如图 15.7 所示。

安全 | https://movie.douban.com/top250?start=25&filter=

图 15.7　豆瓣电影 TOP250 第二页的 URL

可见，URL 中 start 参数用来表示需要显示的电影排名起点。我们只需要修改 start 参数的值，即可获取不同页的内容。为了方便访问，我们将爬取 URL 封装成一个函数：

```
def crawlURL(page):
    print 'Crawling URLs... ... Page : %s' % (page,)
    offset = (page - 1) * 25
    url = 'https://movie.douban.com/top250?start=%s&filter=' % (offset)
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36',
    }
    res = requests.get(url, headers=headers)
    html = res.text
    urls = re.findall(r'(?<=<a href=")https://movie.douban.com/subject/\d+/(?=">)',
html)
    return urls
```

这个函数接受一个 page 参数，并通过“(page-1) * 25”将其转换成 start 参数的值。在函数中，我们使用正则表达式来提取电影详细页面的 URL 地址。我们直接调用这个函数访问第一页并将得到的 URL 列表输出进行测试，可以得到如下输出：

```
Crawling URLs... ... Page : 1
[u'https://movie.douban.com/subject/1292052/',
u'https://movie.douban.com/subject/1291546/',
u'https://movie.douban.com/subject/1295644/',
u'https://movie.douban.com/subject/1292720/',
u'https://movie.douban.com/subject/1292063/',
#以下部分省略......
```

可见，我们成功地获取了电影的 URL 信息。得到 URL 列表后，我们可以以同样的方式编写函数来获取电影的详细信息：

```
def crawlinfo(url):
    print 'Crawling info... ... URL : %s' % (url,)
    global mutex
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36',
    }
```

```
    res = requests.get(url, headers = headers)
    html = res.text
    id = re.search(r'\d+', url).group(0)
    name = re.search(r'(?<=<span property="v:itemreviewed">).*?(?=</span>)', html).
group(0)
    director = re.search(r'(?<=rel="v:directedBy">).*?(?=</a>)', html).group(0)
```

通过查看详细页的代码,我们可以编写对应的正则表达式。本例中,我们只提取了 id、电影名、导演信息。读者可以编写更复杂的正则表达式来提取更多信息。在 HTML 中提取信息,正则表达式不一定是最佳的选择,读者还可以使用 xpath、BeautifulSoup 等工具从 HTML 中提取信息。

完成了提取信息的工作,为了存储信息,我们需要使用数据库。本例我们使用 SQLAlchemy 的 orm 框架与 SQLite 存储信息。首先,我们要声明并创建数据表:

```
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///doubantop250.db', encoding='utf-8', echo=False)

Base = declarative_base()
session_class = sessionmaker(bind=engine)


class Movie(Base):
    __tablename__ = 'movie'
    id = Column(BigInteger, primary_key=True)
    name = Column(String)
    director = Column(String)

    def __init__(self, id, name, director):
        self.id = id
        self.name = name
        self.director = director

    def __str__(self):
        return '<Movie id = %s name = %s director = %s>' % (self.id, self.name.encode
('utf-8'), self.director.encode('utf-8'))

Base.metadata.create_all(engine)
```

运行后,我们将得到一个具有 movie 表的数据库“doubantop250.db”。接下来,我们需要对爬取的内容进行存储。由于网络访问的速度一般远远低于 CPU 的执行速度,为了节省时间,我们可以采用多线程爬虫来爬取页面。这样就相当于同时打开多个网页,节省了依次等待网页加载的时间。

使用多线程会为程序带来一个问题,即线程安全问题,特别是在有数据库操作时。为了

保证线程安全，我们使用互斥，在数据库操作前请求锁，在数据库操作后释放锁。

最后，为了程序使用方便，我们通过接收用户输入来决定执行的功能是创建数据库、爬取信息、还是访问爬取的信息。这个爬虫的代码综合处理后如下：

```
# -*- coding:utf-8 -*-
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.ext.declarative import declarative_base
import re
import requests
import time
import threading

engine = create_engine('sqlite:///doubantop250.db', encoding='utf-8', echo=False)

Base = declarative_base()
session_class = sessionmaker(bind=engine)


class Movie(Base):
    __tablename__ = 'movie'
    id = Column(BigInteger, primary_key=True)
    name = Column(String)
    director = Column(String)

    def __init__(self, id, name, director):
        self.id = id
        self.name = name
        self.director = director

    def __str__(self):
        return '<Movie id = %s name = %s director = %s>' % (
        self.id, self.name.encode('utf-8'), self.director.encode('utf-8'))


def crawlURL(page):
    print 'Crawling URLs......Page : %s' % (page,)
    offset = (page - 1) * 25
    url = 'https://movie.douban.com/top250?start=%s&filter=' % (offset)
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36',
    }
    res = requests.get(url, headers=headers)
    html = res.text
    urls = re.findall(r'(?<=<a href=")https://movie.douban.com/subject/\d+/(?=">)',
html)
```

```
    return urls


def crawlinfo(url, mutex):
    print 'Crawling info... ... URL : %s' % (url,)
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36',
    }
    res = requests.get(url, headers=headers)
    html = res.text
    id = re.search(r'\d+', url).group(0)
    name = re.search(r'(?<=<span property="v:itemreviewed">).*?(?=</span>)', html).
group(0)
    director = re.search(r'(?<=rel="v:directedBy">).*?(?=</a>)', html).group(0)
    mutex.acquire()
    session = session_class()
    session.add(Movie(id, name, director))
    session.commit()
    mutex.release()

op = raw_input('Please choose mode : 1.Create Table 2.Crawl 3.Print Result\n')

if op == '1':
    Base.metadata.create_all(engine)
elif op == '2':
    urllist = []
    for i in range(1, 11):
        urllist += crawlURL(i)
        time.sleep(3)

    mutex = threading.Lock()

    for url in urllist:
        time.sleep(1)
        threading.Thread(target=crawlinfo, args=(url, mutex)).start()

elif op == '3':
    session = session_class()
    movies = session.query(Movie).all()
    for movie in movies:
        print movie
```

在使用时，我们首先使用 1 选项建立数据库。接着，使用 2 选项开始爬取，爬取时的输出如图 15.8 所示。

当爬取结束后，我们使用 3 选项来查看爬取到的内容（Windows 下 cmd 与 PowerShell 默认使用 gbk 编码，而我们使用的是 utf-8 编码，会导致中文输出乱码，我们可以在 cmd 或 PowerShell 中输入“chcp 65001”来切换到 utf-8 编码）如图 15.9 所示。

```
PS H:\python> python .\15-2-1.py
Please choose mode : 1.Create Table 2.Crawl 3.Print Result
2
Crawling URLs... ... Page : 1
Crawling URLs... ... Page : 2
Crawling URLs... ... Page : 3
Crawling URLs... ... Page : 4
Crawling URLs... ... Page : 5
Crawling URLs... ... Page : 6
Crawling URLs... ... Page : 7
Crawling URLs... ... Page : 8
Crawling URLs... ... Page : 9
Crawling URLs... ... Page : 10
Crawling info... ... URL : https://movie.douban.com/subject/1292052/
Crawling info... ... URL : https://movie.douban.com/subject/1291546/
Crawling info... ... URL : https://movie.douban.com/subject/1295644/
Crawling info... ... URL : https://movie.douban.com/subject/1292720/
Crawling info... ... URL : https://movie.douban.com/subject/1292063/
Crawling info... ... URL : https://movie.douban.com/subject/1291561/
Crawling info... ... URL : https://movie.douban.com/subject/1295124/
Crawling info... ... URL : https://movie.douban.com/subject/1292722/
Crawling info... ... URL : https://movie.douban.com/subject/3541415/
Crawling info... ... URL : https://movie.douban.com/subject/2131459/
Crawling info... ... URL : https://movie.douban.com/subject/1292001/
Crawling info... ... URL : https://movie.douban.com/subject/3793023/
Crawling info... ... URL : https://movie.douban.com/subject/3011091/
```

图 15.8　爬取详细内容

```
Windows PowerShell
Active code page: 65001
PS H:\> python .\15-2-1.py
Please choose mode : 1.Create Table 2.Crawl 3.Print Result
3
< Movie id = 1292052 name = 肖申克的救赎 The Shawshank Redemption director = 弗兰克·德拉邦特 >
< Movie id = 1291546 name = 霸王别姬 director = 陈凯歌 >
< Movie id = 1295644 name = 这个杀手不太冷 Léon director = 吕克·贝松 >
< Movie id = 1292720 name = 阿甘正传 Forrest Gump director = 罗伯特·泽米吉斯 >
< Movie id = 1292063 name = 美丽人生 La vita è bella director = 罗伯托·贝尼尼 >
< Movie id = 1291561 name = 千与千寻 千と千尋の神隠し director = 宫崎骏 >
< Movie id = 1295124 name = 辛德勒的名单 Schindler&#39;s List director = 史蒂文·斯皮尔伯格 >
< Movie id = 1292722 name = 泰坦尼克号 Titanic director = 詹姆斯·卡梅隆 >
< Movie id = 3541415 name = 盗梦空间 Inception director = 克里斯托弗·诺兰 >
< Movie id = 2131459 name = 机器人总动员 WALL·E director = 安德鲁·斯坦顿 >
< Movie id = 1292001 name = 海上钢琴师 La leggenda del pianista sull&#39;oceano director = 朱塞佩·托纳多雷 >
< Movie id = 3793023 name = 三傻大闹宝莱坞 3 Idiots director = 拉吉库马尔·希拉尼 >
< Movie id = 3011091 name = 忠犬八公的故事 Hachi: A Dog&#39;s Tale director = 拉斯·霍尔斯道姆 >
< Movie id = 1291549 name = 放牛班的春天 Les choristes director = 克里斯托夫·巴拉蒂 >
< Movie id = 1292213 name = 大话西游之大圣娶亲 西遊記大結局之仙履奇緣 director = 刘镇伟 >
< Movie id = 1291560 name = 龙猫 となりのトトロ director = 宫崎骏 >
< Movie id = 1291841 name = 教父 The Godfather director = 弗朗西斯·福特·科波拉 >
```

图 15.9　爬取到的内容

参 考 文 献

［1］ Wesley Chun. Python 核心编程［M］. 孙波翔，李斌，李晗，译. 北京：人民邮电出版社，2016.
［2］ Allen B Downey. 像计算机科学家一样思考 Python［M］. 赵普明，译. 北京：人民邮电出版社，2013.
［3］ Mark Lutz. Python 学习手册［M］. 侯靖，译. 北京：机械工业出版社，2009.